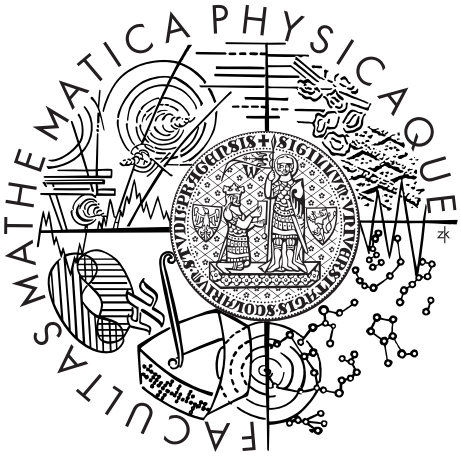
Charles University in Prague

Faculty of Mathematics and Physics

BACHELOR THESIS



Martin Preisler

<martin@preisler.me>

Unified Editor for CEGUI

Department of Software and Computer Science Education

Supervisor of the bachelor thesis: Martin Babka
Study programme: Computer Science
Specialization: Programming

Prague 2012

Many thanks to my supervisor, Mgr. Martin Babka, for helpful advices and mentoring through the development. I would also like to thank contributors who provided feedback, bugfixes and valuable features.

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In _____ date _____

_____

TITLE: Unified Editor for CEGUI

AUTHOR: Martin Preisler

DEPARTMENT: Katedra teoretické informatiky a matematické logiky

SUPERVISOR: Mgr. Martin Babka

ABSTRACT: This thesis presents a free software GUI application called the CEGUI Unified Editor. The application is mainly written in Python and is licensed under GPLv3. Its purpose is to create and modify assets of graphical interfaces made with the CEGUI library. Features include project management, imageset editing and layout editing. Data for older versions of CEGUI are transparently converted using compatibility layers. Big emphasis is put on ease of use, collaboration between multiple content authors and portability.

KEYWORDS: CEGUI, editor, imageset, layout, GUI design

NÁZEV PRÁCE: Unified Editor pro CEGUI

AUTOR: Martin Preisler

KATEDRA: Katedra teoretické informatiky a matematické logiky

VEDOUCÍ BAKALÁŘSKÉ PRÁCE: Mgr. Martin Babka

ABSTRAKT: Cílem této práce je vytvořit GUI aplikaci zvanou CEGUI Unified Editor. Aplikace je z převážné části psaná v Pythonu a licencovaná pod GPLv3. Jejím cílem je tvorba a změna grafických rozhraní realizovaných knihovnou CEGUI. Podporovány jsou mimo jiné management projektů, editace imagesetů a editace layoutů. Data pro starší verze CEGUI jsou převáděna za běhu pomocí vrstev kompatibility. Velký důraz je kladen na jednoduchost používání, možnosti kolaborace mezi více autory a portabilitu.

KLÍČOVÁ SLOVA: CEGUI, editor, imageset, layout, tvorba GUI rozhraní

Contents

Part I

Introduction

Chapter 1

Why editing tools?

There are quite a few CEGUI resources described in Section 2.3. All of them are definitely editable with just text editors. Why am I creating a WYSIWYG editing suite for them then?

The answer is quite simple. While they are all editable and relatively easily understood by programmers and power users, artists will not touch them without WYSIWYG tools. And it is almost impossible to make a nice GUI with just programmers. Furthermore, making small polishing modifications is extremely hard without a visual tool because the developer has to start the application, observe what is wrong, stop the application, edit the data by hand and repeat. The iteration is far too long and makes polishing very time consuming.

This editing tool can bridge the gap between programmers and artists in the team so that they can collaborate to create a working, useful and good looking end product.

Chapter 2

What is CEGUI?

This chapter is focused on understanding key CEGUI concepts and being able to think about GUI development in their terms. Take it as a very quick introduction. Unless stated otherwise, this section is describing CEGUI 1.0.

CEGUI does not invent anything extremely strange when compared to other UI toolkits but there are a couple of areas where your typical UI toolkit will differ a lot from CEGUI. The rest of the document assumes that you have gone through this chapter or already understand CEGUI.

2.1 History

CEGUI has been started by PAUL D. TURNER in 2003 to fill the blank space of advanced GUI subsystems for games and realtime 3D applications. It has seen continuous development since then. For a few years CEGUI [5] was one of the very few open source GUI systems around and quickly became the de-facto standard choice, especially in the Ogre3D [9] community.

Even though the development went over a few bumps, the most notorious being the lead developer suddenly erasing his SourceForge account and leaving the project for a year, the other open source GUI systems have yet to surpass CEGUI when it comes to advanced fine tuned game user interfaces.

2.2 Philosophy

API design

Designed from the ground up to be flexible and extensible. The codebase is written in C++ centred around OOP paradigms and design patterns. Feature patches have to be of high quality to get integrated.

Choice and flexibility

CEGUI is all about choice and flexibility. You can choose a renderer (OpenGL, DirectX9, 10, 11, Ogre, Irrlicht, ...), resource provider (reading plain files is provided as the default resource provider), image codec (FreeImage, SILLY, DevIL, ...), XML parser (expat, Xerces, libxml, ...) and many more. For this reason many believe that CEGUI has many dependencies while in fact it lets you choose the dependencies according to your needs - you can run CEGUI with just OpenGL and the inbuilt default resource provider. Granted, it is not going to be very useful but it proves the point that there are no strictly required dependencies.

2.3 Resources

2.3.1 Imageset

Imageset is a CEGUI-specific term, it is usually called *texture atlas* in other publications and APIs. *Imageset* is composed of a texture (or underlying image) and a set of *image definitions*. Let us look at an example of one such imageset.

```
<Imageset name="OurTestingImageset" imagefile="
   UnderlyingImage.png" version="2">
    <Image name="FirstImage"
           xPos="0" yPos="0"
           width="32" height="32" />
    <Image name="SecondImage"
           xPos="32" yPos="0"
           width="32" height="32" />
</Imageset>
```

Figure 2.1: simple imageset XML example

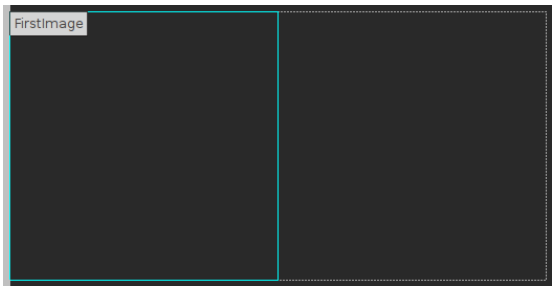


Figure 2.2: visual representation of imageset in Figure 2.1

Let us say the UnderlyingImage.png is an image of size 64x32. The imageset describes two image definitions called *OurTestingImageset/FirstImage* and *OurTestingImageset/SecondImage*, both of 32x32 px in size and both laid out next to each other as seen in Figure 2.2.

You may be wondering why this added complexity? Why not just load two files each containing a different image and store them in two textures? The reason for all this trouble is performance. GPUs are not that fast when it comes to switching textures, the performance hit cannot be ignored on even cutting edge GPUs. Even though CEGUI might just load the different files and texture pack[1] at runtime, it would be a performance hit when starting the application and that is not acceptable.

Fortunately you can get the best of both worlds and use offline texture packing offered by *ceed-mic*. This makes maintenance easier while still retaining all of the performance benefits. And the texture packing process can be slow and precise because you only do it once and then load the resulting data over and over again quickly.

Since version 1.0 CEGUI no longer has the *Imageset* class in its API. Imageset is just a file format to comfortably describe multiple image definitions on one texture. The API only knows about the resulting images, not their relationship.

2.3.2 Font

Allows developers to describe a font in CEGUI context. If the *freetype2* dependency has been met it can be a TTF font. Or it can list character to image mappings to form a pixmap font[2].

```
1 <Font
2     name="DejaVuSans-10"
3     filename="DejaVuSans.ttf"
4     type="FreeType" Size="10"/>
```

Figure 2.3: TTF example

[1] Making a texture atlas from them automatically.
[2] Pixmap font is also called a bitmap font.

```
<Font
    name="FairChar"
    filename="FairChar.imageset"
    type="Pixmap" version="3">

        <Mapping codepoint="65" image="A" />
        <Mapping codepoint="66" image="B" />
        <Mapping codepoint="67" image="C" />
</Font>
```

Figure 2.4: pixmap font example

It is important to note here that different sizes of a font have to be defined multiple times in CEGUI. For example if we wanted DejaVuSans in sizes 10 and 12, we would define DejaVuSans-10 and DejaVuSans-12 fonts separately. The system rasterises vector glyphs of each font and puts them onto one or more textures. Each texture used for a font has a multitude of rasterised glyphs on it. When CEGUI is told to draw "Hello world!" with given font, it generates a quad for each glyph and uses UV-coordinates of that particular glyph.

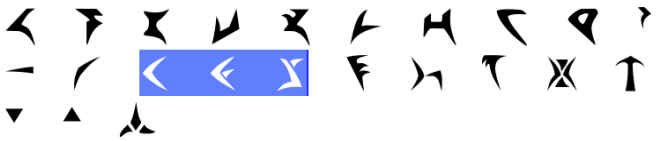


Figure 2.5: CEGUI drawing unicode text

There is bidirectional font support but that is out of scope of this document.

2.3.3 Layout

The main goal of layout files is to eliminate maintenance pain when dealing with complex UI hierarchies. These are XML files containing necessary info to construct a widget hierarchy, much like .ui files in Qt or .glade files in Gtk. Even though it is possible to write layout files manually in a text editor, they are very suitable for WYSIWYG editing.

```
<GUILayout version="4">
    <Window Name="root" Type="DefaultWindow">
        <Window Name="child" Type="TaharezLook/
           StaticText">
            <Property Name="Position" Value="{{0,0},
               {0,0}}" />
            <Property Name="Size" Value="{{0.5,0},
               {0.5,0}}" />
        </Window>
    </Window>
</GUILayout>
```

Figure 2.6: XML layout representing a static text child window taking one quarter of the space of its parent

It is important to understand what layouts actually result in after loading. *WindowManager* returns a single *Window* pointer that contains everything that was loaded from the layout. You can use the usual means of the getChild, isChild, ... methods to explore and use the result.

```
CEGUI::Window* layout = WindowManager::getSingleton().
   loadWindowLayout("SomeLayout.layout");
CEGUI::Window* child = layout->getChild("child");
```

Figure 2.7: querying a child window

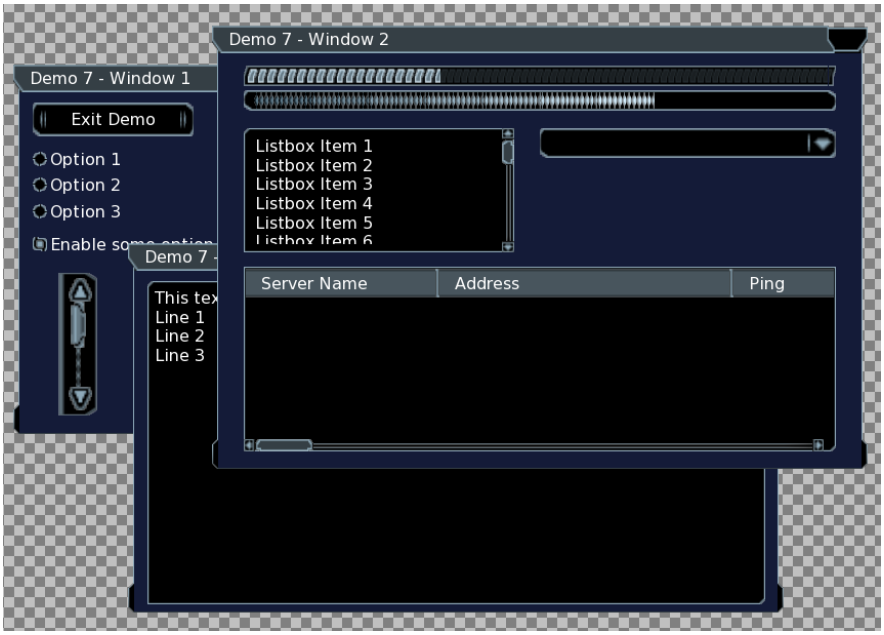


Figure 2.8: example of a complex GUI layout

```
<Animations>
    <AnimationDefinition autoStart="false" duration="
       0.3" name="Example1A" replayMode="once">
        <Affector applicationMethod="absolute"
           interpolator="float" property="Alpha">
            <!-- progression of the first keyframe will
                be ignored -->
            <KeyFrame position="0" progression="linear"
                value="1" />
            <KeyFrame position="0.264" progression="
               quadratic decelerating" value="0.66" />
        </Affector>
    </AnimationDefinition>
</Animations>
```

Figure 2.9: XML animation example

2.3.4 Animation

Widgets (and several other classes) in CEGUI use a string-based introspection implemented by *CEGUI::PropertySet*. Animations in CEGUI interpolate values of these properties using a chosen *interpolator*. The interpolator always acts on two neighbour keyframes that are next to the position of the animation. Each keyframe has a *progression*[3] describing whether the animation towards it is linear, accelerating or decelerating.

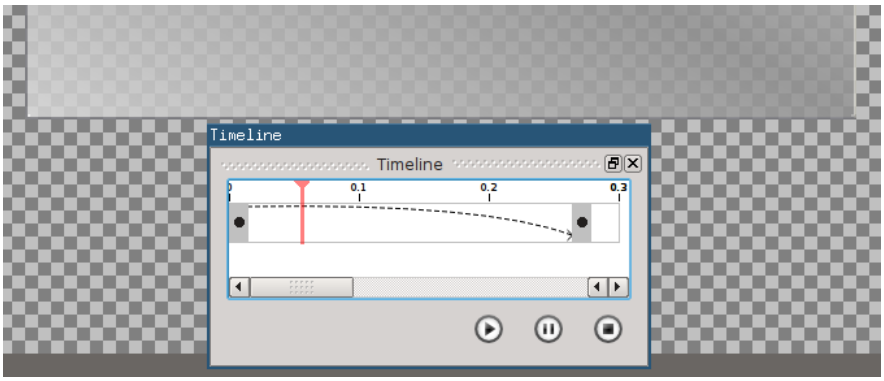


Figure 2.10: visual representation animation in Figure 2.9

[3]Progression is a CEGUI specific term, it is also known as “easing” in other software.

2.3.5 Scheme

```
<GUIScheme version="5" name="AlfiskoSkin">
    <Font filename="DejaVuSans-10.font"/>
    <Imageset filename="AlfiskoSkin.imageset"/>
    <LookNFeel filename="AlfiskoSkin.looknfeel"/>
    <WindowRendererSet filename="
       CEGUICoreWindowRendererSet"/>

    <FalagardMapping windowType="AlfiskoSkin/Label"
       targetType="DefaultWindow" renderer="Core/
       Default" lookNFeel="AlfiskoSkin/Label"/>
    <FalagardMapping windowType="AlfiskoSkin/Button"
       targetType="CEGUI/PushButton" renderer="Core/
       Button" lookNFeel="AlfiskoSkin/Button"/>
    <FalagardMapping windowType="AlfiskoSkin/
       ImageButton" targetType="CEGUI/PushButton"
       renderer="Core/Button" lookNFeel="AlfiskoSkin/
       ImageButton"/>
    <FalagardMapping windowType="AlfiskoSkin/
       RadioButton" targetType="CEGUI/RadioButton"
       renderer="Core/ToggleButton" lookNFeel="
       AlfiskoSkin/RadioButton"/>
    <FalagardMapping windowType="AlfiskoSkin/Checkbox"
       targetType="CEGUI/ToggleButton" renderer="Core/
       ToggleButton" lookNFeel="AlfiskoSkin/Checkbox"/>
</GUIScheme>
```

Figure 2.11: short XML scheme example

Scheme is what describes how and in which order CEGUI should load resources. It also defines widgets by mapping widget type, *LookNFeel* and possibly *RenderEffect*. In a way it is the glue that connects and holds all your UI work together. Unlike several other GUI toolkits, CEGUI developers believe that you should always know which assets you want loaded and you should be in complete control of that. For example images are never loaded automatically.

2.4 Widgets

2.4.1 Foundation

CEGUI widgets are build on top of functionality from two other classes.

Element

CEGUI::Element most notably implements positioning and sizing. Each element has a list of child elements and represents a node in a tree graph. All elements created form a forest graph. For most cases we will only consider the subtree of the element we are working with.

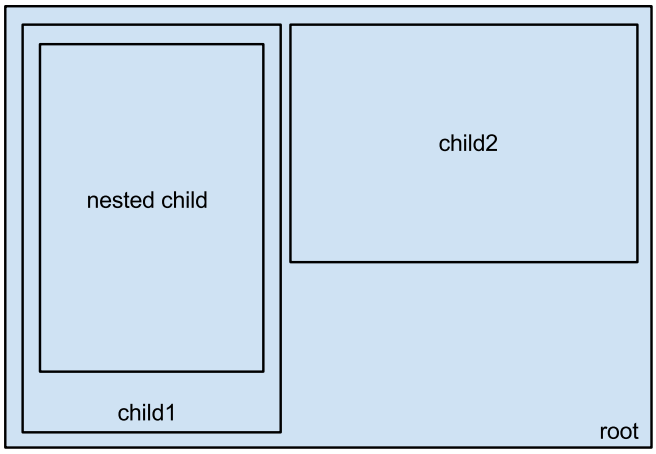


Figure 2.12: element hierarchy

Keep in mind that Elements by themselves do not have names, the element hierarchy shown in Figure 2.12 has names inside for demonstration purposes and to make it easier to compare it to named element hierarchy shown in Figure 2.13.

NamedElement

An extension of the Element class called NamedElement adds names and name paths to the system. All widgets are inherited from the NamedElement class which means all widgets have names and you can query them with name paths. Widgets' names are unique only in the context of their parent so two widgets with exactly the same name can coexist peacefully. If you add them both to the same parent you will get an exception though.

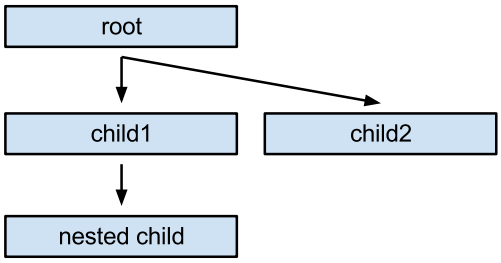


Figure 2.13: named element hierarchy

Name path is a series of named element names separated by "/". It can be compared to filesystem relative path. Let us consider the named element hierarchy shown in Figure 2.13. Element *nested child* can be retrieved from root by calling:

```
root->getChildElement("child1/nested child");
```

2.4.2 Widgets are named elements

CEGUI::Window, also known as "widget", inherits from *CEGUI::NamedElement* which inherits from *CEGUI::Element*. It therefore has traits of both Element and NamedElement. More features are added on top including *LookNFeel* skinning, *RenderEffects*, input processing, ...

It is common to call a tree of widgets a *GUI layout*.

2.4.3 Every tree starts with a root

CEGUI will only render and allow interaction with widgets that are attached to a particular widget tree. In 0.7 and earlier versions this widget tree started with a widget called *GUI sheet*, since 1.0 it is possible to have multiple of these trees and therefore multiple user interfaces (even projected onto objects in the 3D world). Widgets not attached to these trees are simply disregarded when input events are handled and are not being rendered unless custom code does that. The different user interface roots are called *GUI contexts*. The class implementing these is *CEGUI::GUIContext*. One widget can only be the root of one GUI context, no widgets can be shared between GUI contexts.

2.4.4 Unified dimensions (UDim)

In older CEGUI versions (before 0.4), there were 2 types of dimensions in the system - absolute and relative. In version 0.4 they were merged into a single dimension unit called *UDim*. *UDim* is simply an aggregate of the two dimension types. Merging two dimension types together allowed greater power and flexibility in the system and better code maintainability.

Relative (scale)

Represents length relative to the parent's size (width or height, depending on what the UDim represents). For example horizontal 0.5 with a parent of width 100px results in 50px. Very useful in many cases, allows resolution independent layouts, especially if combined with aspect ratio locking.

Absolute (offset)

This is an absolute offset by given amount of pixels, independent of any other variable - always results in the same amount of pixels. Should be very familiar to most users as it is used in many other widget toolkits.

Combining it all together (UDim)

By taking numbers representing both relative (scale) and absolute (offset) dimensions and storing them in one class we get UDim. You can think of UDim as a way to describe length, it is used both for positions and sizes. It is important to note that the relative component comes first.

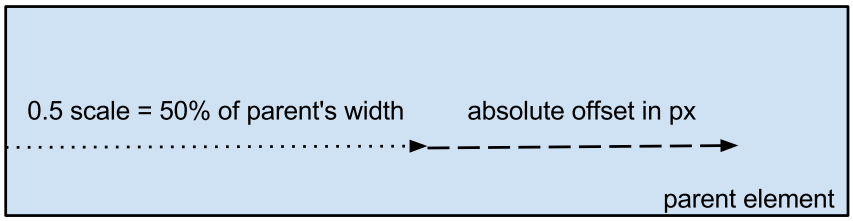


Figure 2.14: visualisation of horizontal UDim

$$abs(x) = abs(p) * x_{scale} + x_{offset}$$

Where p is parent's size, if widget has no parent 0 is considered as parent's size.

Figure 2.15: converting UDim to absolute pixels

Examples

Let us say we have a widget called *Parent* and a child inside that widget called *Child*. For the purpose of this section the *Parent* has absolute width of 100px and absolute height of 200px.

```
1 child.setPosition(UVector2(UDim(0, 0), UDim(0, 0)));
2 // 0% of parent's size and 100px on top of that = 100px
3 child.setSize(USize(UDim(0, 100), UDim(0, 100)));
```

Figure 2.16: top-left child, 100x100 px

```
1 child.setPosition(UVector2(UDim(0.5, 0), UDim(0, 0)));
```

Figure 2.17: centre-left child

This approach centres the widget's left edge but it will not centre it as such. For this to happen we have to subtract half the widget's size as seen in the following example.

In the next example we want our child to be centred horizontally, completely at the bottom vertically and have absolute width of 50px and absolute height of 50px.

```
1 // relative puts it at the centre, absolute moves it
     back half its size
2 child.setPosition(UVector2(UDim(0.5, -25), UDim(1.0,
     -50)));
3 // this is a no-brainer, 50x50px
4 child.setSize(USize(UDim(0, 50),  UDim(0, 50)));
```

Figure 2.18: centre-bottom child, 50x50 px

Keep in mind that you are recommended to use CEGUI's alignment properties, you could use just *UDims* but your code would be less readable and maintainable. See Section 2.4.5.

2.4.5 Positioning

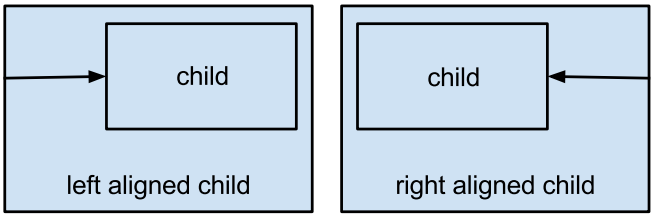


Figure 2.19: alignment and position (represented by arrows)

Alignment

Despite showing you how to do alignment with just UDims, doing so using the horizontal and vertical alignment properties is more practical and easier. Alignment basically defines what the base widget position will be, the *Position* property then offsets relative to that base position. In the default settings, alignment is set to *top left*, so the *Position* property works as expected from its name. By aligning a widget *top right*, its right edge will be touching the parent's child content area right edge (if horizontal *Position* is {0, 0}). It is much easier

to position a widget and then deal with its size, with our previous approach we would have to keep altering both the position and size UDims.

Offsetting with the "Position" property

After you have decided which alignment the widget should have, you can offset it to fine tune its position. Choosing the correct alignment first will save a lot of time.

Z-order

CEGUI has to know whether currently rendered widget should overlap any of its siblings. To decide this, each window has a draw list of child widgets in order of their rendering[4]. Clicking a widget will usually "raise" it in the render list.

2.4.6 Sizing

There are two types of sizing in CEGUI 1.0 and previous versions - widget sizing and Falagard sizing. The latter is considerably more powerful and offers operators to perform calculations on sizes.

Widgets

Size of widgets is a simple vector of two UDims. See Section 2.4.4. You cannot perform any dynamic calculations directly but you can set the relative component to make child's size relative to its parent.

Falagard

The Falagard skinning/LookNFeel system offers a more powerful version of sizing, apart from just specifying relative and absolute size, it allows you to add, subtract and perform other trivial operations with a set of values of your choice. You can reference various sizes or hardcode constants. The structure to hold the equation is a binary evaluation tree.

[4]Widgets getting rendered later overlap widgets rendered previously.

Part II

Development

Chapter 3

Planning phase

3.1 Previous CEGUI tools

There have been many attempts and even successful implementations of various tools around CEGUI. *CELayoutEditor* and *CEImagesetEditor* were supported all the way until CEGUI 0.7 (including). Since then "CEED" (The CEGUI Unified Editor) is the official swiss-knife type tool for all CEGUI resources. I will briefly mention the older tools, mainly because a lot of their concepts were transferred straight into CEED and because I believe it makes sense to mention them because they served well and for quite a long time. It is undeniable that CEED would not have looked the same if it were not for the previous tools.

3.1.1 CEImagesetEditor

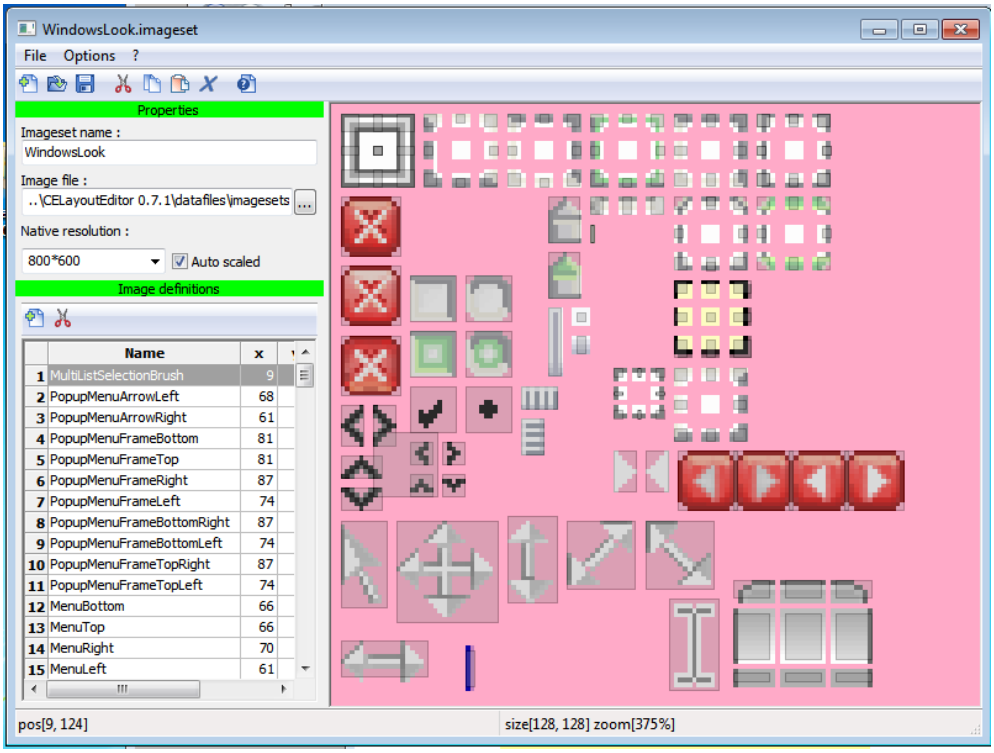


Figure 3.1: screenshot of CEImagesetEditor 0.7.1

Written originally by MARTIN FLEURENT [19] using wxWidgets. Offers simple imageset editing without offset support. The most painful disadvantage in my opinion is the lack of undo/redo which makes delicate editing quite hard. Making mistakes by accidentally drag-moving is disastrous and the only solution is to reload from the last saved state. Initial commit has happened in January 2005 and it is still used in 2012, although abandoned and obsoleted by CEED.

I have been greatly inspired by this tool, most of the core concepts of imageset editing in CEED are taken directly from *CEImagesetEditor*.

3.1.2 CELayoutEditor

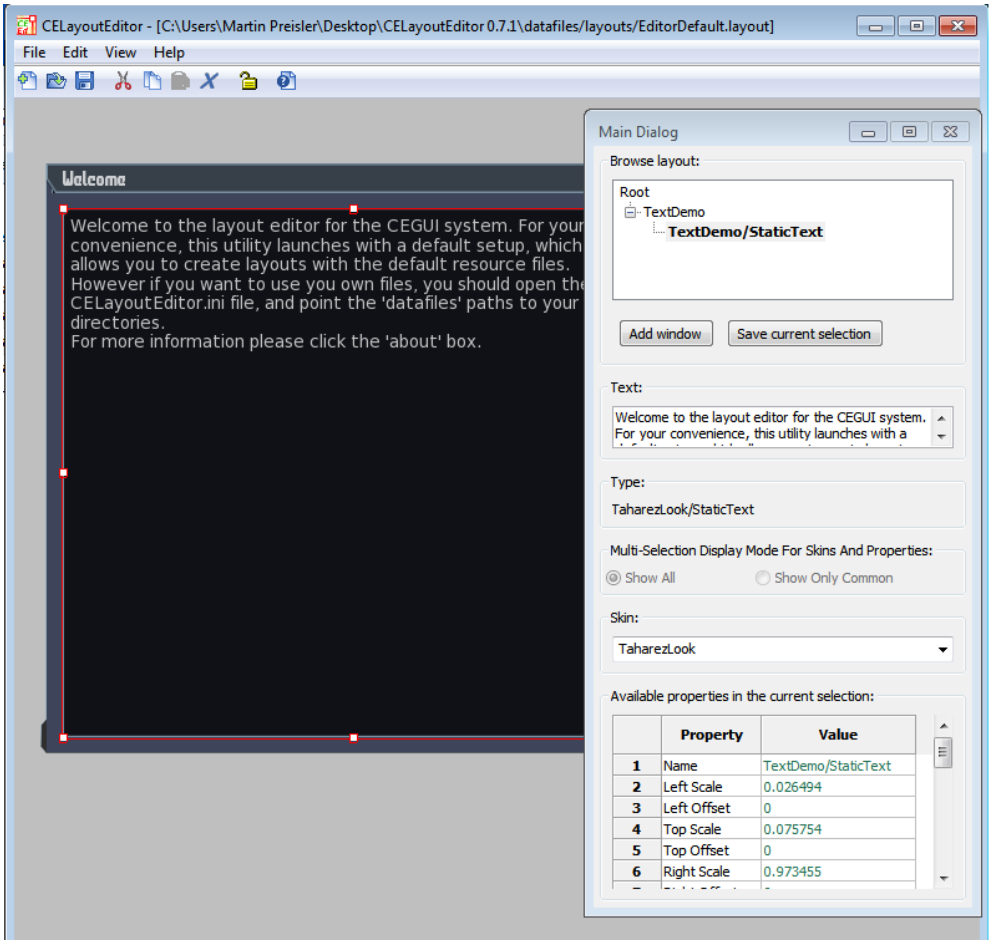


Figure 3.2: screenshot of CELayoutEditor 0.7.1a

Written originally by PARTICK KOOMAN [20] using wxWidgets, uses CEGUI internally to draw the layout. Has been supported all the way until 0.7 (including). In my experience it is not very robust but it is stable if you get all the inputs just right.

Again, this was a source of inspiration, many concepts of layout editing in CEED draw directly from this application.

When compared to CEED it has no undo/redo which makes it frustrating to the point of being unusable for many people. It also lacks compatibility layers, multi selection, widget hierarchy copy paste and code editing.

3.2 Related non-CEGUI tools

3.2.1 Qt designer

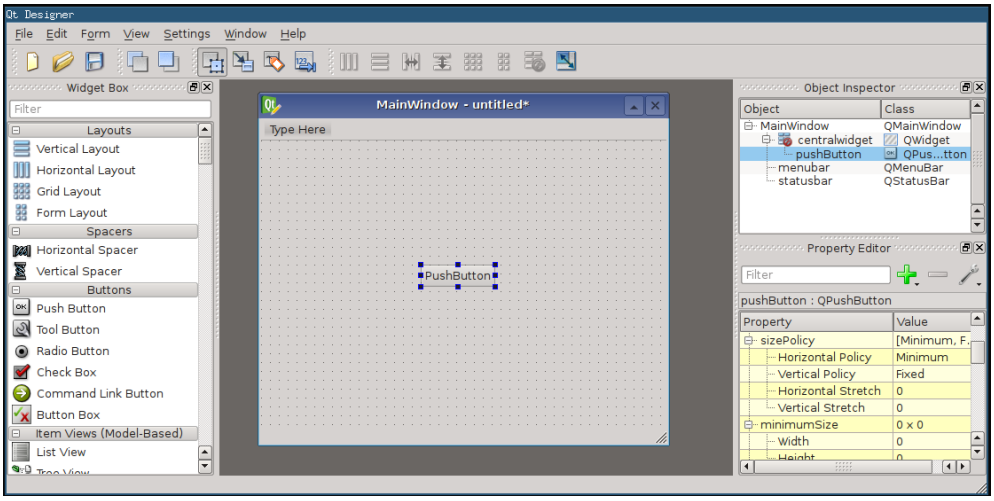


Figure 3.3: screenshot of Qt Designer 4.8.2

The official RAD tool from the Qt project [21]. Offers very advanced GUI layout editing. This application was used to design GUI layouts for CEED and its layout editing has been heavily inspired by it.

Instead of the multi-tab paradigm, layouts are all shown at once as separate root widgets.

3.2.2 Glade

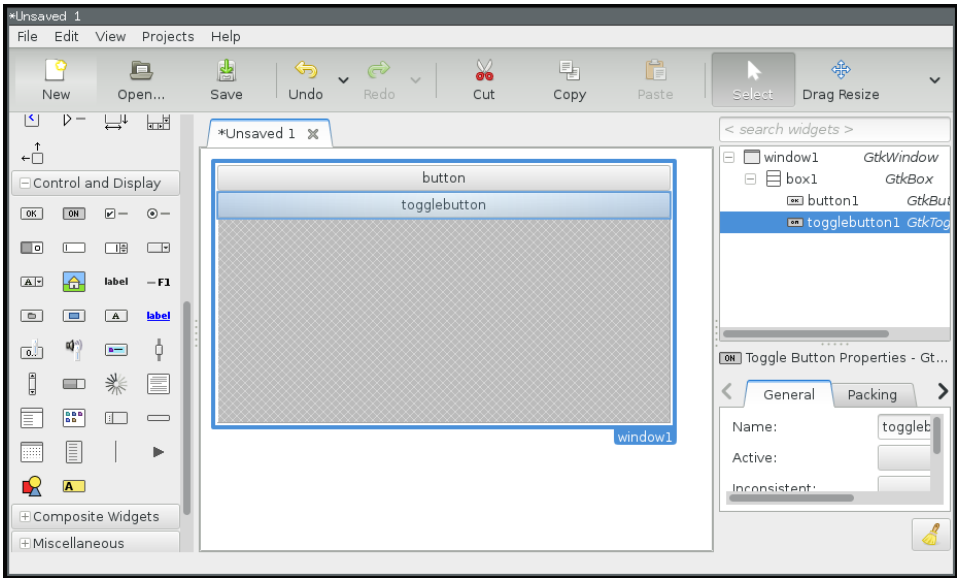


Figure 3.4: screenshot of Gtk Glade 3.12.1

Equivalent of the Qt designer in the Gtk world. Allows editing of multiple files in separate tabs. Despite finding it clunky and unpredictable when doing certain actions it has inspired CEED in many ways.

3.2.3 MyGUI Layout Editor

MyGUI can be thought of as a library directly competing with CEGUI. It has similar goals and similar features [8].

A difference that stands out is that the layout editor is using MyGUI for its controls and you are editing a MyGUI layout inside it. I found that a bit awkward and it makes the tool very disconnected with the rest of running applications. I am not sure what the reasoning was but I cannot see any benefits of it, except maybe to demonstrate that MyGUI can be used for complex applications.

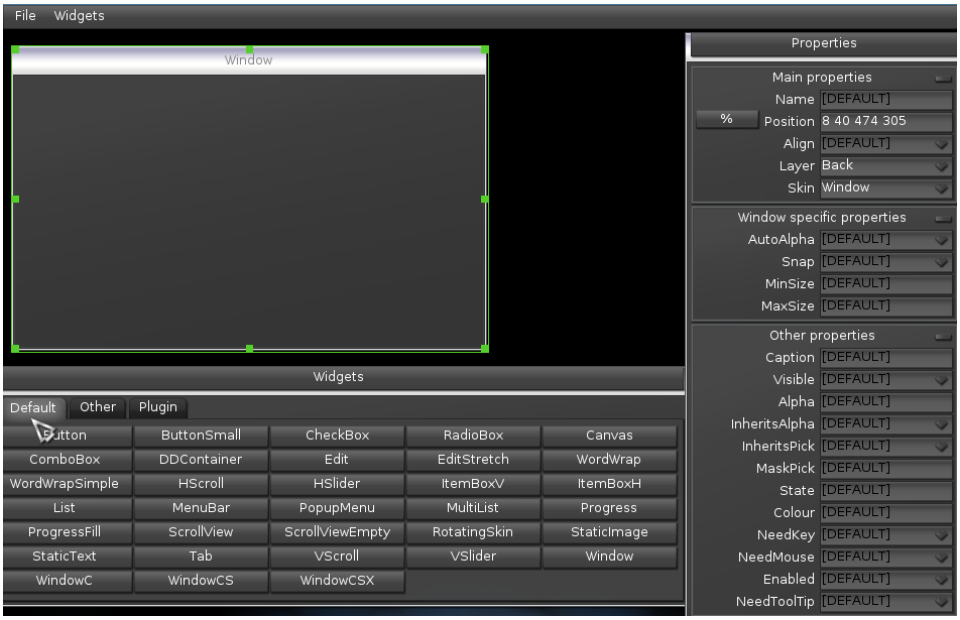


Figure 3.5: screenshot of MyGUI layout editor

There is the familiar property editor and a create new widget panel. Widget manipulators look and feel a lot like Qt Designer's.

3.3 Design goals

3.3.1 Target audience

I imagine the typical user as an artistic, creative person, with strong background in image editing tools, with some knowledge of XML but next to no knowledge of CEGUI internals.

This drives all the requirements. I try to make the tool fit this model user. This proved to be very hard, especially as I do not fit this role at all. I have no background with image editing tools, I am a power user who uses terminal emulators more than anything and I know quite some CEGUI internals.

3.3.2 Cornerstone requirements

Before I started the project I had some general broad design goals I wanted to accomplish no matter what. You can take them as cornerstones of CEED, no part of the application can violate them. The reason why I obsess about these few rules is that the previous tools violated them and it made them arguable painful to work with. I did not want to improve the situation slightly, I wanted to push the bar to another level.

Free software (GPLv3+)

I knew from the start that this the project is too big to be done alone and that I would need extensive testing through the development. As I am a big fan of free software this was a fairly easy decision.

The pain of NDAs, EULAs and so on would mitigate all potential monetisation. Python applications are also very hard to close, decompiling Python bytecode is trivial. I could have obfuscated the bytecode for each release but again, it is a pain and yet another added complexity. Bug reports and backtraces would also be very hard to collect.

Cross platform

As an enthusiastic GNU/Linux user I know the pain of Windows-only applications. I wanted to support at least GNU/Linux, Windows and MacOSX. I have made deployments on other platforms possible, all used technologies are very portable.

Easy to use and intuitive, yet powerful

These are contradicting goals, the plan was to make a reasonable compromise. I decided to offer a flat learning curve with "verbose" GUI but offer shortcut customisation for actions. When I could not satisfy both goals I preferred ease of use, taking the target audience as described in Section 3.3.1 into account.

Multi document interface (implemented as tabs)

This was a trade-off between usability and reliability. There is no need to start multiple applications to edit multiple files. On the other hand, any sort of crash can affect all opened files and all unsaved changes can be lost in the worst case.

Undo redo for all actions on all files (undo stacks split by file)

Something the community repeatedly requested in the previous tools for years. In my opinion this is a must in any productivity application. The user must not be afraid of losing progress at any point.

Ongoing compatibility support via an extensible API

Allows people to start working on a project targeting one CEGUI version and then migrating to a new version later in the process. Enables smooth, almost seamless migration of files. Another motivation for this requirement was to gain adoption quicker to gather more bug reports. Most projects were and still are at this point (22nd July 2012) using CEGUI 0.7. Talking them into trying a tool that is not useful for their project is very hard.

Workflow based around projects, not files

I noticed that artists were struggling with setting up paths and other variables to have CEGUI start up properly in the previous tools. My vision was to make a reusable, relocatable project configuration file that can be committed into a source control repository and shared by all team members. An experienced member in the team can set all the paths and others just pull the changes and load the project.

Another reason was that the previous tools had global settings for paths which allowed the user to work on only one project at a time. Switching projects meant going through the paths and setting them to respectable values - a very time consuming process.

3.3.3 Relaxed requirements

These goals would be nice to have but are not strictly necessary.

Unify CEGUI asset editing tools

CEED can provide a foundation other community members can build editing tools on. Eliminate the initial cost of building up embedded CEGUI facilities,

settings interfaces, action interfaces, etc.

Having support for all assets CEGUI can use in CEED would be great but is very hard to achieve and is a moving target.

Integrate breakpad or other means of automatic bug reporting

This has not been implemented as of 22nd July.

The rationale is that the target group are artists, they are not usually proficient at reporting technical bugs manually. Supporting a way to automatically submit backtraces and reports of crashes would be very useful to the developers.

Chapter 4

Programming phase

4.1 Technologies used

Python 2

Chosen for a rich standard library, being well known and having a large community. It is really fast to develop with but in my opinion requires discipline to avoid ending up with unmaintainable code. Bindings for Qt [11] and CEGUI are available. It is obviously slower than languages that compile into native code but that is not an issue for a content editing tool.

OpenGL

The only viable option because DirectX is not cross-platform. It is easy to embed and well supported.

Qt 4

A proven, mature and reliable GUI toolkit. Has a very useful RAD tool called *Qt Designer*.

CEGUI

The only viable choice to render CEGUI layouts without reimplementing parts of CEGUI in Python.

cx_Freeze

Used to prevent the need for users to compile the dependencies themselves. Allows standalone "frozen" binaries that work without any dependencies and do not need to be installed system-wide.

4.2 Tools

4.2.1 Version control

I used *Mercurial* for all version tracking since it allows collaboration with many authors without the merging pain that is so prevalent in *Subversion* and *CVS*.

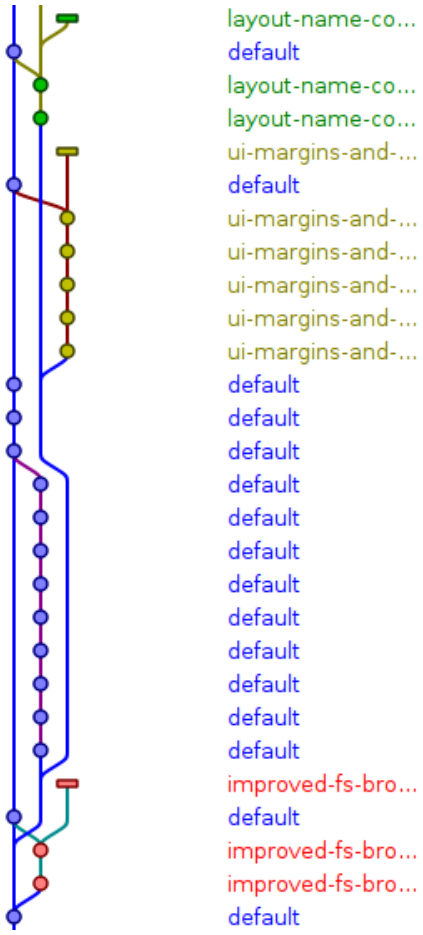


Figure 4.1: use of Mercurial in CEED

4.2.2 Code editing

Vim and *Eclipse* were the tools that I wrote most of the code in. *Vim* provided insufficient Python autocompletion support and *ropevim* [12] was not very stable when development started. In the end I used *vrapper* which provided vim-like controls inside *Eclipse*. *Eclipse* provided convenient auto-completion and *Pylint* integration.

4.2.3 Bug tracking and planning

Mantis [7] was used for all bug tracking and feature planning. See Section 13.4 for statistics.

4.2.4 Static analysis

Python proved to be extremely quick to develop with but also made it very hard to pull off any extensive refactoring. I ended up with broken snapshot releases that contained trivial to fix but also easy to reproduce show-stopper bugs. In early 2012 I found out that you can find a lot of bugs with static analysis. Unless the code contains monkey patching and abuse of duck typing the static analysis tools can prevent bugs in releases. Since the usage of wildcard imports makes it harder for these tools so I decided to get rid of all wildcard imports and continuously scan the entire code base with pylint and pyflakes.

Many issues that would otherwise require extensive testing to be found were fixed this way.

4.3 Initial stages

The preliminary design was done mostly during February 2011. The idea was brewing in my head a bit earlier though. I started a forum thread to brainstorm making an editor on 9th December 2010 [10].

First commit was pushed on Sunday, 5th March 2011. The development time allocated for the stable "thesis" version was roughly 16 months, until July 2012. My plan was to develop iteratively, avoid cathedral development and employ agile development practices.

4.4 Engaging the community

I knew from the start that this project will need community contributions to survive in the long run, so I tried to embrace contributions. Even though I wrote the overwhelming majority of the code, I value all patches, bug reports and suggestions given to me.

4.4.1 List of contributors

- JORGE AVILA - art, user interface advices
- MARTIN BABKA - supervision of the thesis, advices
- ENIKO - ported rectangle packing code to Python
- MARKUS EWALD - author of the rectangle packing code used in ceed-mic
- MICHAEL KAPELKO - bugfixes for Python 2.6
- CHARLES MATTEI - art, user interface advices
- LUKAS MEINDL - Windows QA, feedback
- ERIK OGENVIK - testing, feedback
- ADAM PREISLER - testing, feedback
- MARTIN PREISLER - main developer
- PATRICK ROBERTSON - settings interface improvements
- PAVEL ŠPAČEK - QA, testing, feedback
- STEFAN STAMMBERGER - author of recently used menus
- PAVLOS TOUBOULIDIS - property inspector and user interface improvements
- CHRIS TRENKAMP - file monitoring, better error messages when files cannot be opened
- PAUL D. TURNER - MacOS X fixes and releases

Figure 4.2: contributor list

See Figure 4.2 that lists contributors in alphabetical order with their contributions roughly listed. The list has been assembled on 28th July. For more info about contributions of a particular person, please run:

```
1 hg log --user Contributor
```

in the repository of the project.

I would like to thank all of the contributors, the project would be nearly impossible without their valuable inputs. See Section 13.2.2 for a breakdown of contributions.

4.5 Release early, release often

Even though I was not confident that the tool could be usable for production in the initial stages, I knew from previous experience that I had to make releases. Having many soft deadlines forced me to do a lot more planning and kept me doing continuous development - many small steps leading to the final goal. I named all releases "snapshotX" where X was the number of the release. This reinforced the fact that they were just developer snapshots and not intended for production[1]. I kept doing standalone win32 binaries for snapshots because I noticed majority of my users are using Windows and many of them could not get the editor running because of dependency problems...

4.6 Early adopters

My goal was to get as much adoption as possible to give me more bug reports and therefore improve quality. That was the reason why I sometimes worked on features that were not outlined in the requirements but were requested by early adopters. Making early adopters happy helped spread the word which brought even more adopters.

The editor has been used by at least three teams working on commercial games. Both teams supplied amazing feedback and bug reports, I would like to thank them for that! I will not list the names because I am not sure whether the games have been announced yet.

Several open source teams adopted the editor in production as well, the most notable being Summoning Wars [13] and The Worldforge Project [14]. They provided data I could test with which helped me make the layout editor usable for creating 0.7 compatible assets.

[1] The snapshots were used for production though, as mentioned in the following section.

Part III

User Manual

Chapter 5

Prerequisites

5.1 Hardware and software requirements

Operating system:

- GNU/LINUX system with X11
- WINDOWS XP and newer
- APPLE MACOS X 10.6+
- *BSD system with X11[1]

Hardware:

- At least Intel Pentium 4 or AMD Athlon XP
- 512 MB of RAM
- Graphics card capable of accelerated OpenGL with FBO support
- Display area of at least 800*600 px available to the application[2]

Dependencies:

- PYTHON 2.7
- QT 4.7 or 4.8
- PYSIDE – python bindings for QT
- CEGUI 1.0[3]
- PYCEGUI 1.0

[1] CEED has been reported to work on BSD but this platform is not officially supported.
[2] Vertical screen estate of at least 800 px is recommended.
[3] CEED supports CEGUI 0.7 data as well but uses 1.0 internally.

5.2 Knowledge prerequisites

To use the Unified Editor, you have to have some basic knowledge of how GUI systems work in general. It would be best if you were familiar with how CEGUI works but Qt or GTK knowledge can be transferred without problems.

For editing assets in *code mode* you should understand the format in question, the editor makes no effort to make code editing easier with highlighting or any code references (it's a planned feature though).

There is no need to understand programming in general to use CEED efficiently, the application is targeted at artists. You may still need help from a fellow programmer about setting up the project file.

5.3 Installation

The application is currently distributed as a source tarball that works on all supported platforms (if you install the dependencies). Furthermore, binary standalone builds are made for Windows and Apple MacOS X. This only applies to official releases, mercurial code is not being built regularly.

5.3.1 Source tarball

First, make sure you have all dependencies installed. Refer to guides of the dependencies on how to install them.

Download the tarball and extract it. You should end up with a folder called *CEED-${version}*. Go into this folder in CLI[4] and call *python setup.py install* as administrator[5]. This should trigger the installation. After the installation finishes, run *ceed-gui* to start the application.

5.3.2 Standalone executable (Win32)

You do not have to install anything in this case (all dependencies are bundled), just unzipping the archive and running *ceed-gui.exe* will make the application start. If this is not the case, please report it as a bug.

If you have made custom changes to CEGUI that you use with your application, this distribution method might not work well for you! Upstream CEGUI is used for the build.

[4]Use terminal emulator of choice on UNIXes, cmd.exe on Windows.

[5]Use su/sudo on UNIXes, "Run as administrator" on Windows.

5.3.3 .app bundle (MacOS X)

CEED behaves like a proper MacOS X native application. Double clicking will start the GUI, dragging it to *Applications* will install it. If you use multiple versions they might share settings!

If you have made custom changes to CEGUI that you use with your application, this distribution method might not work well for you! Upstream CEGUI is used for the build.

Chapter 6

Working with the application

6.1 The basics

6.1.1 Main interface

This interface hosts all the tabbed editors and provides functionality that is shared by all the editors. It surrounds the application.

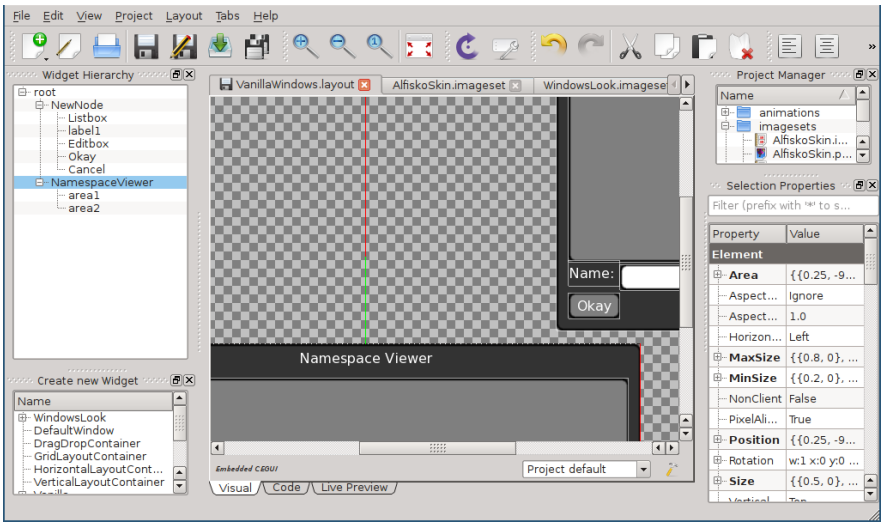


Figure 6.1: overview of the interface

6.1.2 Multi tab editing

The centre of the application consists of tabs, each tab represents one opened file. CEED will strive not to have two tabs for one file opened, if you try to open one file twice it will just activate the existing tab for the file. Any of the tabs can be closed on request. If there are unsaved changes the user will be asked what to do about them.

Reordering tabs is also possible using mouse drag-moving. The interface including dock widgets, toolbars, etc... may change when switching tabs. Switching tabs does not count as an undo action, it is instead just a context change action.

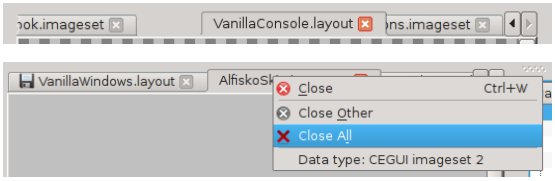


Figure 6.2: tabs offer reordering and a context menu

6.1.3 Multi mode editing

For some file types (mostly .imageset and .layout) it makes sense to edit in both visual mode as well as code mode (raw XML). Undo and redo are transparent between mode switching as mode switching counts as an undo action. You can switch modes by clicking on the bottom tab pane. All files are opened in Visual mode by default. Code editing is regarded as a crutch mechanism, a tool to make tedious mass changes or work around problems with Visual mode.

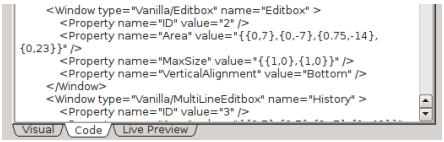


Figure 6.3: switch modes by clicking the bottom mode tabs

Code editing is currently very simplistic and does not even have syntax highlighting! Please also note undo/redo is very wasteful when it comes to code editing and long editing sessions may end up with a lot of RAM being allocated.

6.1.4 Copy / Paste

Many things you can select in the editor can be copied/cut and pasted elsewhere. This is a very useful workflow feature and works even across multiple CEED instances. You can for example have 2 different projects and copy parts of layouts between them. Both projects have to have all widget types that or copied of course!

The default shortcuts are the expected ones depending on your platform. *Ctrl* + *C* for copy on Windows and GNU/Linux, *Cmd* + *C* on Mac OS X.

6.1.5 Project manager

Allows user to manage a project (project is a set of related files), browse through files, open any file for editing, add new or existing files to the project and remove files from the project. Most of the project managing takes place in the *Project Manager* dock widget. The dock widgets lists all files currently in the project and allows user to edit/view them by double clicking them. Right clicking brings up a context menu with the ability to add files to the project or remove currently selected files from the project. The main reason for project management is to have CEGUI resource path settings shared for all the files in the project. The project files are designed to be committed to a repository and used on different computers. All paths are stored relative to the project file, even though you may see absolute paths in the editor itself, they get converted to relative paths in the end.

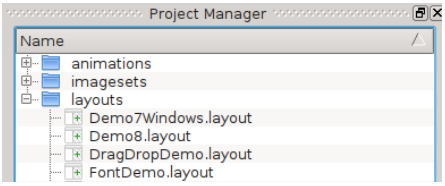


Figure 6.4: project manager dock widget

6.1.6 File manager

You are advised to use project manager if at all possible, it will improve workflow, especially in bigger teams.

In addition to the project manager, CEED allows you to browse the filesystem and simply open files for quick editing. Please note that this will only work for some file types (it cannot work for schemes and layouts because CEGUI paths are not set if project is not loaded). This is again contained in a dock widget that hosts the functionality. It has a very simplistic interface, displaying current path, allowing the user to go one level up the hierarchy and simply listing files and folder in the current path.

Double clicking a file opens it for editing.

Furthermore, this dock widget will watch for changes on the filesystem and refresh accordingly.

6.1.7 Resizable rectangle

A construct used reused in many places in CEED. Represents a rectangular object that can be resized and/or resized. It offers features that would otherwise have to be reimplemented in many places. The main inspiration was GIMP's selection rectangle [6]. Can be moved by dragging the middle area. There are eight areas that resize the rectangle if you drag them. These are called resize

handles - four edges and four corners. They get highlighted when you hover over them.

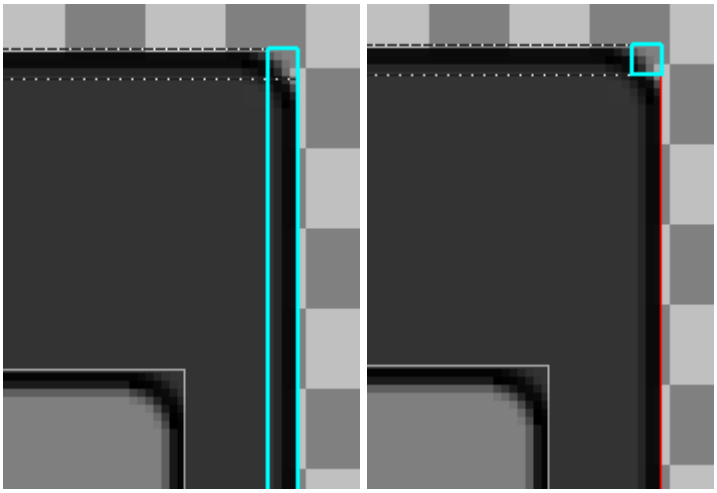


Figure 6.5: resizable handles appear when you hover over edges/corners

6.1.8 Zooming

To make editing easier and possibly more precise, user is allowed to zoom in the editors that support it. Both imageset and layout editors support zooming. By default you can zoom by clicking the zoom icons in the toolbar as seen in Figure 6.1 or holding the *Ctrl* key and scrolling with your *mouse wheel*. You can zoom out to 50% to get an overview or zoom in multiples of two. If you are at the zoom level of 100%, one zoom in will change that to 200%, another zoom in will change it to 400%, etc.

The zoom level will in no way affect your data, it will only change the context you are editing in. It therefore is not an undo action.

6.1.9 Undo and Redo functionality

Since most tabbed editors allow undo and redo, the main interface allows you to perform these actions via shared means in the top toolbar. Undo/redo are per-file, so whenever you switch tabs the undo stack gets changed to a completely different stack. If you for example do changes to file *A*, then switch to file *B*, do some changes and keep pressing *undo*, only changes to file *B* gets undone, You have to switch to *A* again and undo there. Selections are not undoable as they do not count as undo actions. There is one very important aspect that breaks the "context switching is not an undo action" rule. Switching editing mode is an undoable action because the changes of XML code do not make sense in visual mode.

Undo and Redo can be very powerful and allows free experimentation without fear of losing data. Apart from possible bugs in the applications, all things that affect data are undo actions and are undoable. Use that fact to your advantage!

Even with undo/redo, you are advised to use version control or other mechanisms. Undo/redo actions are lost when you close the application!

6.1.10 Compatibility layers

CEED has a facility called compatibility layers which allows it to work with data from many versions of CEGUI. In a nutshell these layers allow you to transform raw data from version to version. Each layer has source and target data types that describe the transformation. Layers can be chained, if there is a path to go from data type *A* to *B* and *B* to *C*, it is possible to transform *A* to *C*.

Each time you are editing a file, you can right click the tab to bring out a context menu and see the exact data type you are editing, see Figure 6.2. It is not currently possible to change the data type of a file that's been opened, CEED will attempt to detect the data type when you are opening the file and ask you if it cannot decide for sure. Make sure you set the correct CEGUI target version as described in Figure 6.7. The CEGUI version will affect what data type newly created files in CEED will have.

You can also use all compatibility layers using the shipped CLI tool called *ceed-migrate*. Call *ceed-migrate –help* for more info.

6.2 Creating a project

6.2.1 Creating a project file

You will need to create a project for any serious editing with CEED. There are some quick hints about how to do this in the *Quickstart Guide* but we will dig into more details here.

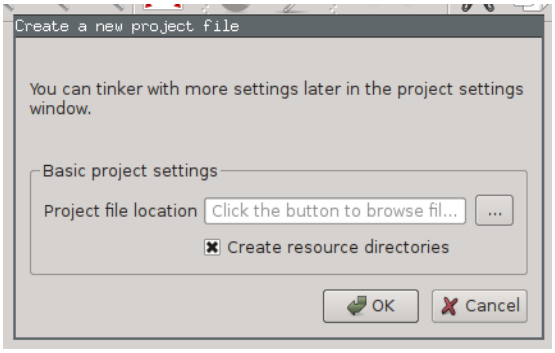


Figure 6.6: first step of creating a project

The first step is to choose *File* » *New* » *Project*. The dialog in Figure 6.6 will appear. The most important choice you have to make is where to store the project file. It is recommended to store it in a directory made exclusively for the project. Doing so ensures that it is relocatable. You can optionally instruct CEED to auto create resource directories similar to the CEGUI sample datafiles directory structure.

6.2.2 Project settings

Project settings window will pop up after the project file is created. You can always return to the project settings window to change settings later. It is recommended to get the basics right when creating the project as that will prevent many headaches[1]. Most of the options are documented in the interface as can be seen on the following screenshots.

[1] CEED uses the project to choose versions of new files for example.

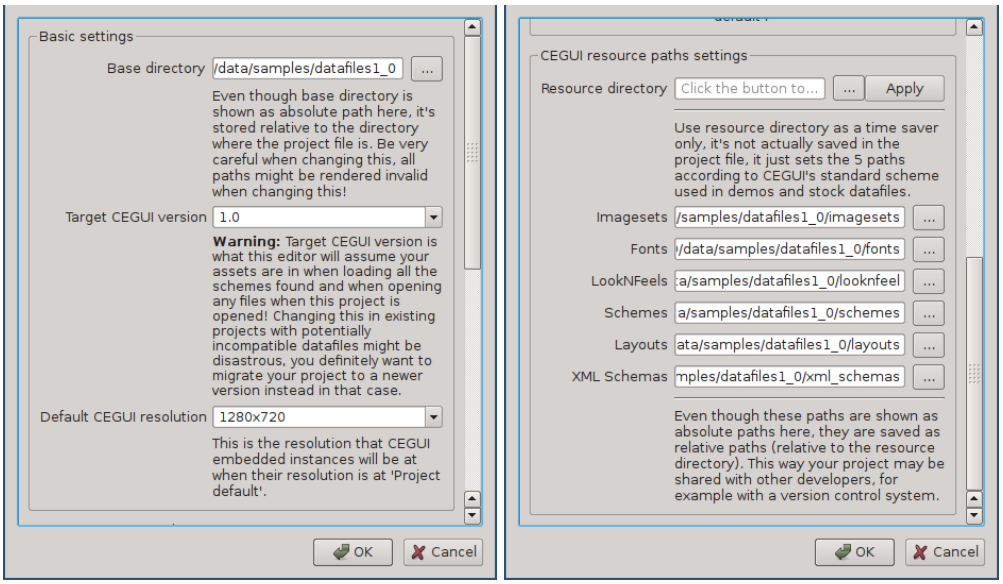


Figure 6.7: project settings window

All paths stores in the project file are relative to the parent directory of the project file. This allows you to move the project directory around and share it with your co-developers. First thing you need to choose is the base directory. This will by default be the same as the folder your project file is in. You can however set it to the base of your CEGUI resources and the rest of CEGUI resource paths will be relative to it. This allows to switch paths around quickly. If you are unsure, just leave it as it is.

The next thing you need to choose is the target CEGUI version. CEED supports CEGUI 1.0 and 0.7. This has very drastic consequences, especially on any resource files you create from scratch, so make sure you set it correctly.

Following is the section with paths to CEGUI resource directories. If you use CEGUI's directory structure (the one used in samples), you can simply fill or browse for the first editbox and press apply. Otherwise you will have to fill the resource paths manually. Even though the paths are shown as absolute, they are stored *relative* to the project's base directory, which again is stored relative to parent directory of the .project file. This is what makes the whole project directory relocatable.

6.3 Imageset editing

An imageset is practically a texture atlas, a technique used to put many smaller images on one texture to drastically speed up rendering. Each imageset has a name, underlying texture file path, autoscaled settings, native resolution and a set of images. “Image” means a rectangle selection of the underlying texture. Each of these images has a position (x, y), width, height and an offset (x, y).

Opening existing imagesets for editing does not require you to have a project opened. The underlying texture file is sought after in the same directory the .imageset file resides in. If a project file is opened the appropriate resource directory is used to search for the files instead.

6.3.1 Overview

A dock widget containing basic imageset properties and a list of image definitions shows up upon opening an imageset file. The centre part of the editor shows the underlying texture and rectangles describing geometry of image definitions. Image definitions can be selected (including multiple selection). Selection works in both the image definition list and the editor’s centre part (rubber band selection). Both selections are synchronised. Names of image definitions will be shown as labels when the definitions are selected. This can be disabled in settings, see Section 6.6 for more details.

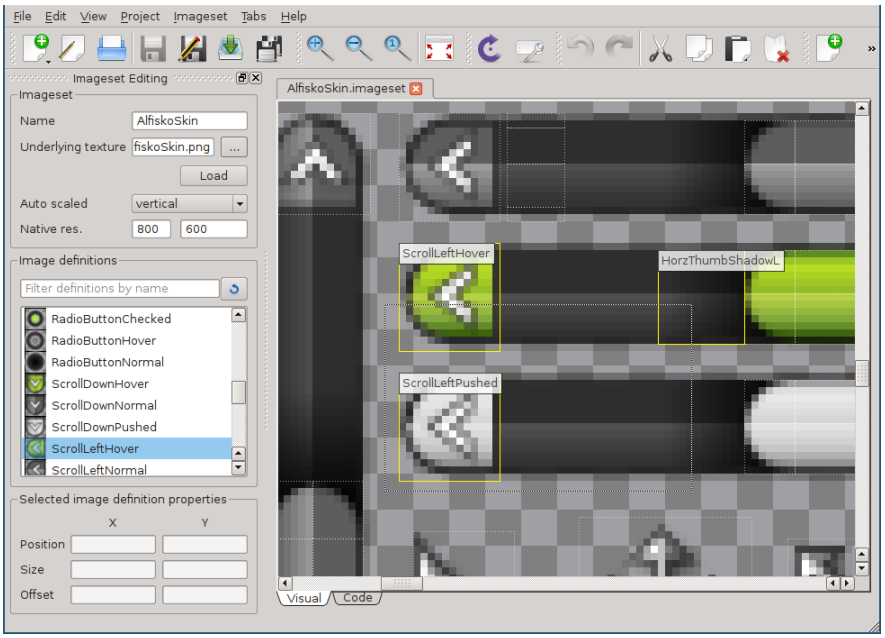


Figure 6.8: rubber band selecting in the imageset editor

6.3.2 Imageset properties

As can be seen in Figure 6.8, imageset itself has several properties that affect all image definitions defined in it.

The name of the imageset means different things depending on which version of CEGUI you are targeting. It is simply a prefix of each of the defined image definitions' names in CEGUI 1.0 with "ImagesetName/ImageName" being the full name of a single image definition, in 0.7 image definitions are strictly tied to imagesets and are referenced using "set: ImagesetName image: ImageName".

Underlying texture is the texture that will be used to draw the imagery of image definitions inside the imageset.

The auto scaled settings affects how the imageset's image definition sizes will be affected on various resolutions. In a nutshell, with disabled auto scaling the image definitions will always have exactly the size they are defined with, with any other auto scaled settings, the size will be affected by the resolution CEGUI is currently rendered in (see CEGUI documentation for explanations of each setting). Please note that only "false" and "true" settings are implemented in CEGUI 0.7, the rest are new in CEGUI 1.0.

Native resolution is the resolution this imageset was created for in simplified terms. If the target resolution is different, the image definitions may be scaled, depending on the *auto scaled* settings.

6.3.3 Moving and resizing image definitions

You can move selected image definitions by dragging or using keyboard (*WSAD* scheme moves them one pixel in each direction, pressing *Ctrl* moves them 10 pixels in each of the directions). You can also resize the image definitions by pressing *Shift* and one of the *WSAD* keys, this moves just the bottom right *vertex* of the image definition rectangle. Rectangles are "resizable rectangles", see Section 6.1.7 for more details. This will only delete the rectangle definitions of the images in the imageset, it will not alter the underlying texture in any way.

6.3.4 Deleting image definitions

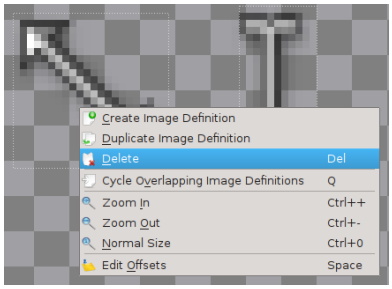


Figure 6.9: selected images context menu

You can delete a selection of image definitions by selecting, right clicking and choosing *Delete* in the context menu or pressing the *Delete key*. This will only delete the image definitions, it will not alter the underlying texture in any way.

6.3.5 The property box

To allow precise adjustments, user can alter all the values of selected image definition manually using the property box. Just select exactly one image definition and the property box will get filled with its values. Altering them will immediately preview the changes in the visual editing pane. All editing has undo/redo support, see Section 6.1.9 for more details. Editing image definition properties in the property box should always be preferred to editing raw XML in code mode, it is safer and has better and faster undo/redo.

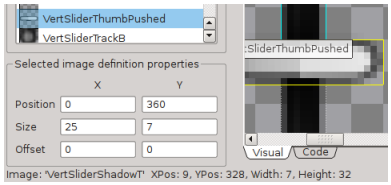


Figure 6.10: image property box

6.3.6 Editing image definition offsets

Offsets describe where the *0, 0* position is in the image definition. By default it is at the top left corner of the rectangle of the image definition. You may want to adjust it, especially if you are working with crosshairs, cursors or window edge imagery. Enable editing of offsets using the context menu (see Figure 6.9) and red offset crosshairs will appear, move them around to alter the offsets. You can alternatively use the property box to edit offsets manually using the keyboard, see Section 6.3.5.

6.3.7 Selecting overlapping image definitions

Sometimes the image definitions overlap in a way that would prevent you from selecting the image definition you need. You can select any image definition in the overlapping area and use the *Cycle Overlapping Images* feature. It will cycle all overlapping image definitions, just stop when the desired one is selected. The order in which the overlapping image definitions are cycled is not well defined, just cycle until the right one is selected. You can see the option in the context menu in Figure 6.9.

6.3.8 The code mode

Raw editing of the XML is supported in imageset editing, click the "Code" tab in the bottom to switch to it. Please note that the raw format will always be the native CEGUI 1.0 format even if you selected CEGUI 0.7 as the target version. The conversion will only happen when you save to a file.

6.4 Layout editing

The word layout has two meanings in the CEGUI world so I will first disambiguate it. The first meaning is a hierarchy of widgets, the layout describes the way widgets are laid out as an *n-tree*. The other meaning is a widget able to lay out its child widgets during runtime, this is more precisely called *Layout Container*. I will use the word "layout" with the former meaning in the rest of this section unless stated otherwise.
Each layout starts with a root widget and can only have exactly one root widget. If you need more top widgets, just put them into a *DefaultWindow* and use the *DefaultWindow* as the root widget. When CEGUI loads the layout it returns this root widget. Widgets always have a name, this name only needs to be unique in the parent in CEGUI 1.0, it has to be globally unique in CEGUI 0.7. Saying "widget Parent/Child/SubChild" means a widget that is a child of the "Child" widget and the "Child" widget is a child of the "Parent" widget. This notation is called the *name path*.

6.4.1 Overview

Upon opening a layout, dock widgets containing widget hierarchy, available widgets to create and selection properties show up. The centre part of the editor shows the layout as rendered using CEGUI. Widgets can be selected and unselected by clicking on them, rubber band selection is not available in the layout editor. Coloured lines are shown to outline how the widgets are sized and positioned. Properties of selected widget(s) are shown and are available for editing in the bottom right part of the screen.

There are no global properties of the GUI layout, the only objects holding information are the widgets.

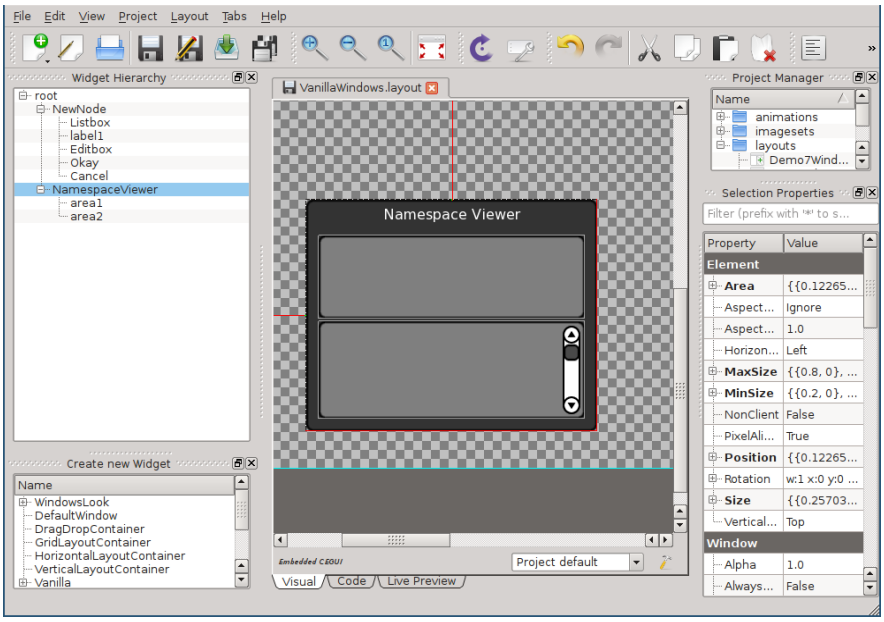


Figure 6.11: selecting a window when layout editing

6.4.2 Moving and sizing widgets

Selected widgets can be moved around by dragging. Resizable rectangle is used for implementation of the *widget manipulators*, see Section 6.1.7 for more details.

Unified dimensions

Both position and size have two components in CEGUI - scale and offset. Scale represents the dimension relative to the parent, offset is absolute pixels. Both can be positive or negative, both are stored as floating point numbers.

There are several tools in CEED to deal with unified dimensions without having to edit them manually in the property editor (see Section 6.4.4). By default CEED affects the offset component, meaning you are moving everything in pixels. It is recommended to learn relative positioning and sizing, it is a very powerful tool to make resolution independent layouts. Press the *A key* to switch which dimension component you are affecting with your moving and resizing.

Figure 6.12: unified dimensions of a button

Use *Normalise Position* and *Normalise Size* to "clean up" the dimensions in your layout. Default shortcuts are the *D key* and *S key* respectively.

Whenever you select a widget, its position and size are shown by the means

of coloured lines. Red part of the dimension represents the scale component and green part represents the offset component.

Snap to grid

It is really hard to keep your widgets aligned and it is quite easy to spot for your users that they do not align properly. CEED has a tool to help you align your widgets to a grid. You can enable it by pressing the *Spacebar key* or by clicking the *Snap to grid* icon. If it is activated, a grid will be shown whenever you drag move any widgets and your movements will snap to it. All snap to grid happens only between parent and its child widgets, so the grid will be only shown in the dragged widget's parent.

You can change snap grid settings in CEED's application settings, see Section 6.6.

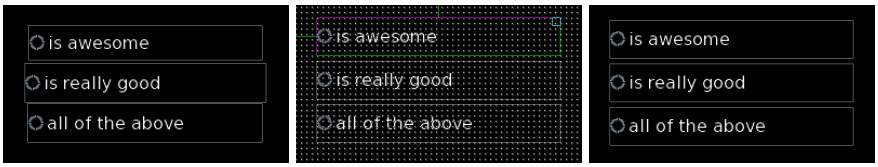


Figure 6.13: snapping to grid, zoom at 200%

Alignment

Position of a widget in CEGUI is relative to a certain pivot. By default this pivot is the top left corner of the widget's parent widget client area. You can change this by selecting a different alignment. Choosing *Centre* for both horizontal and vertical alignments.

Figure 6.14: alignment of widgets

Minimum and maximum size

Another tool to help you create resolution independent layouts. Consider you are using relative sizes and your top with is 50% of the screen width. You can place a limit that the widget should be 50% of the screen width but not exceed 1000px by setting the MaxSize property to (0, 1000).

The maximum and minimum sizes are unified dimensions but there is one big difference: The first component is relative to the screen dimension, not to the widget's parent!

Aspect ratio

This feature was added in CEGUI 1.0 and it is not available in earlier versions. The compatibility layer will simply strip it and the behavior will be as if the AspectMode was set to Ignore.

Is a tool to make sure your UI is not stretched in any way in a case where your display's aspect ratio differs from that of your user. Choose the desired AspectMode first using the property editing (see Section 6.4.4). There are three modes you can select: *Ignore*, *Shrink* or *Expand*. *Ignore* ignores your chosen aspect ratio and does not affect the sizes in any way. *Shrink* makes sure to comply to the aspect ratio by shrinking one of the dimensions until the ratio is satisfied. *Expand* behaves similarly but expands one of the dimensions until the ratio is satisfied.

The dimension guides will reflect this by either drawing the size guide too long or too short for *shrink* and *expand* respectively.

Figure 6.15: aspect ratio demonstration

6.4.3 Deleting widgets

Selected widgets can be deleted by right clicking them in the widget hierarchy dock widget to bring the context menu and choosing *Delete* or by simply selecting them and pressing the *Delete key*. Please note that deleting a widget also deletes all its descendants - children, children of children, ...

6.4.4 Property editing

A well implemented CEGUI widget should expose most of its properties using the *CEGUI::PropertySet* class. All properties exposed as such will be editable within the editor in the *Selection Properties* dock widget.

Simply select a widget, scroll through its properties in the dock widget, choose one and double click it to alter it. Such action will be recorded as an *undo action* and will be undoable, see Section 6.1.9 for more details. You can use the property filter to quickly find the property you are looking for by name, see Figure 6.16.

Some of the properties will show an expandable icon on the left, these are usually complex properties made up from multiple pieces. Clicking this icon will expand the property and allow you to edit its components.

It is possible to edit properties of multiple widgets but properties will display <multiple values> if there is no single value in all of them for that property.

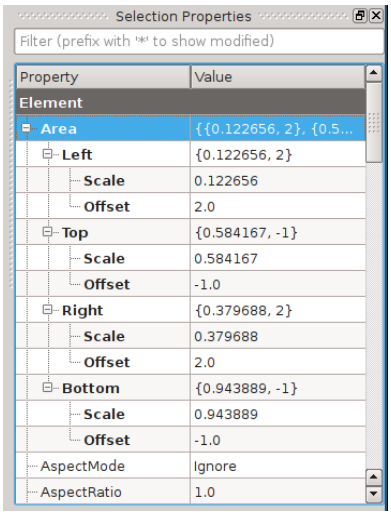


Figure 6.16: recursively expanded Area property

6.4.5 Reparenting widgets

Reparenting means changing parent of a widget or an entire hierarchy of widget. This can be done in the widget hierarchy dock widget, pick a widget there, drag it and drop it into a widget that should be its new parent.

6.4.6 Live preview

You can preview your widget layout including limited interaction by clicking the *Live preview* tab in the bottom. It is possible to then interact with your GUI as if it were in your application. Nothing you do in *Live preview* will affect the edited layout.

6.4.7 Custom widgets

The editor loads all schemes in the project so standard widgets get registered if you have them in your scheme. The scheme will be able to load a custom widget set module. The custom widget set module is able to add widget factories so you can even add custom made widgets and have the editor edit them.

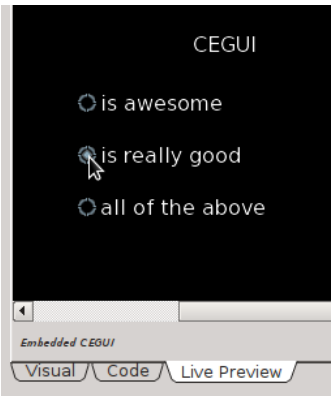


Figure 6.17: previewing a layout

6.4.8 The code mode

Raw XML editing is supported in layout editing, click the *Code* tab in the bottom to switch to it. Please note that the raw format will always be the native CEGUI 1.0 format even if you selected CEGUI 0.7 as the target version. The conversion will only happen when you save to a file.

Please note that there are several errors that you can make in the layout XML that will go silently ignored! This is an issue which CEED cannot solve, it is CEGUI that ignores this input. Examples include various property values, for example the *ColourRect*, invalid values will result in black.

XML editing should only be used as the last resort!

6.5 Command line

While most of this manual talks about the main executable - ceed-gui - two more are offered in all releases. As with most command line tools, inbuilt help is offered and it would make no sense to repeat it here. Instead, a brief overview of some capabilities of the tools will be presented.

6.5.1 ceed-gui

The main use of this executable is to bring up the CEED GUI interface. Most people will probably call it with no command line arguments at all. Some of the supported arguments might be very useful though, you can for example call ceed-gui so that it immediately loads a certain project and opens specific files.

Call `ceed-gui --help` for more info.

6.5.2 ceed-migrate

Allows you to use compatibility layers as described in Section 6.1.10 without having to start the main GUI app. This can be very useful when you want to migrate all your assets to a new CEGUI version. You can also incorporate it to your workflow if you have to support multiple CEGUI versions for some reason.

Call `ceed-migrate --help` for more info.

6.5.3 ceed-mic

Makes it possible to build a rectangle packed imageset out of given data. Source data is a *MetaImageset*. In a nutshell you can specify separate image files or even imagesets and ceed-mic will make just one imageset with one underlying texture out of them. It will make effort to have the resulting texture as small as possible.

See *data/samples/AllStockImagery.meta-imageset* for an example.

Call `ceed-mic --help` for more info.

6.6 Settings

Go to *Edit » Settings* to open the settings window. Note that these settings are only persistent on the user's machine, they are not stored in a project file! If you make changes that you want your colleagues to share, you will have to tell them how to replicate these changes on their machines.

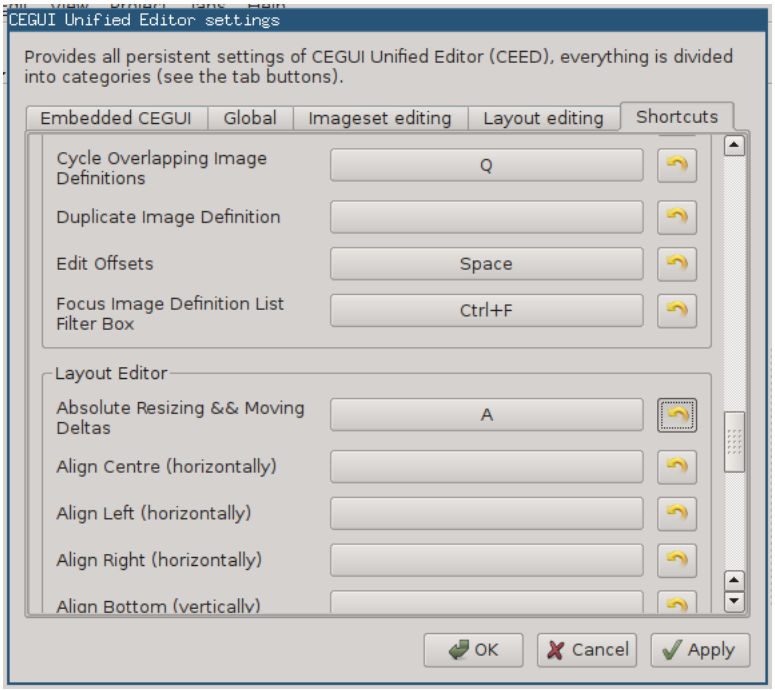


Figure 6.18: settings window

6.6.1 Applying changes

Most of the settings will apply immediately after pressing the Apply button. Some will require you to close and open the edited file again and some will even require you to restart the application. If you experience glitches related to a settings change a restart will very likely cure them. We would still appreciate it if you reported them as a bug though, for further detail see Section 7.3.1.

6.6.2 Back to default

Feel free to change settings and experiment with them. Whenever you regret making some setting, click the *Reset icon* as seen on Figure 6.18 to reset it to its default value. Please note that same rules about applicability of the settings apply when you reset them to default. In some cases you will even have to restart the application.

6.6.3 Shortcuts

Most of the actions in CEED are using the *Action API* and can be triggered using shortcuts. The default shortcuts are set depending on your platform. Go to *Edit » Settings* and choosing *Shortcuts* in the tab header. Clicking the button in the middle will allow you to press your desired combination to change to it. Only combination keypresses are supported, preferably with modifier keys. Discrete sequences or key chords cannot be used for shortcuts.

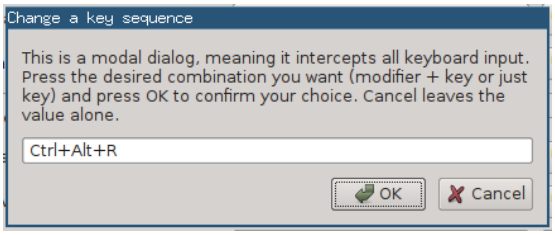


Figure 6.19: changing key sequence for an action

Chapter 7

Further help

7.1 Common issues

Cannot find PyCEGUI

Assuming you are using a release source tarball, make sure you have installed PyCEGUI system-wide. Check the PyCEGUI.{so,dll} with *ldd -r* or *depends.exe* to check whether it can find all required libraries.

You can also use the runwrapper.sh script if you use a UNIX system but that is not officially supported.

Cannot find XYZ

Install dependency XYZ, all dependency upstreams offer binary installers with the exception of PyCEGUI. Check with *ldd -r* or *depends.exe* to see whether the libraries can find their dependencies.

Only widget outlines move when I pan the layout editing viewport

Confirm that zooming also only moves the outlines.

Make sure your GPU supports FBO and that it supports it properly. The solution is to get better OpenGL drivers.

This may also be a newly introduced bug that may be worth reporting. Use common sense.

OpenGL invalid enum exception

Make sure your GPU supports FBO. Other than that it is usually a driver bug. I have witnessed this in VirtualBox VM. The solution is to get better OpenGL

drivers.

This may also be a newly introduced bug that may be worth reporting. Use common sense.

Cannot start 'hg'

This is just a warning and you need not be concerned with it. Mercurial is used to get the *tip* revision hash for error reporting purposes. It will simply "unknown" in cases where it is not possible to get the hash.

I only see a black viewport with outlines instead of my widgets!

Make sure the widgets are set as visible! Many people create layouts with widgets hidden and then show them selectively after they are loaded. CEED will not display hidden widgets, it will only display their outlines.

Imageset editing is unresponsive

There are two configurable modes of drawing the imageset editing viewport. The default one uses OpenGL and redraws the entire viewport when any redraw is needed. It has a predictable speed and does not stall often. It is suitable for all but the biggest imagesets.

In cases where you edit really big imagesets and/or have a slow GPU you can use the alternative mode. It uses software QPainter and only redraws portions that need it. This makes it a bit unpredictable with occasional stalls but it will be faster if you edit in a small area.

The settings option is *Settings » Imageset editing » Use partial drawing updates*.

See Section 6.6 to learn how to alter settings.

Changing sizing modes while resizing/moving widgets

7.2 Getting support

Community support is provided on the following channels. No official commercial support is offered.

- forum: `http://cegui.org.uk/phpBB2`
- IRC: #cegui on irc.freenode.net[1]
- wiki: `http://cegui.org.uk/wiki`

[1] See `http://freenode.net` for information about the FreeNode network.

7.3 Help CEED

7.3.1 Report bugs

Software gets complex really quickly, nobody is pretending that CEED does not have any bugs. If you find some we would really appreciate you reporting them using the CEGUI bug tracker[2]. Please make sure the bug has not been reported already by searching the bug tracker before submitting the ticket. Feature requests are also welcome.

Please note that there are no assurances on whether and when particular tickets get resolved, we do our "best effort" to fix tickets but cannot possibly give you a reliable time frame.

7.3.2 Help with documentation

Documenting is a very hard task, any help is welcome in that area. The usual starting point is editing the CEGUI wiki[3]. If you feel up to it you can also send patches to the manuals, they are written in *LyX* so it should not be a problem to correct typos or even add new content.

7.3.3 Help with development

CEED is a community project, *GPLv3+* licensed. That means that anyone can contribute! See the *Developer manual* for more info.

7.3.4 Donate money

If you would like to help us spend more time developing the software you use and hopefully like, consider making a donation. Every amount counts, no matter how small.

Donating to the main developer of CEED is possible at

`http://sourceforge.net/donate/?user_id=559904`

You can also donate to the CEGUI library itself using

`http://sourceforge.net/project/project_donations.php?group_id=93749`

Both types of donations require a PayPal account.

[2]We have a Mantis tracker at `http://cegui.org.uk/mantis`.
[3]Wiki of the project is at `http://cegui.org.uk/wiki`.

Part IV

Developer Manual

Chapter 8

Prerequisites

8.1 Knowledge requirements

Because of size constrains, I will not cover Python, PySide, Qt and CEGUI API.

8.2 Getting the source code

```
$ hg clone http://crayzedsgui.hg.sourceforge.net:8000/
  hgroot/crayzedsgui/CEED
```

8.2.1 Branches and Tags

- *default* - unstable forward development, likely to be based on unstable CEGUI
- *snapshotX* - development snapshots, based on unstable CEGUI, should be considered tech previews
- **-devel* - feature branches, are expected to be closed and merged into default at some point

8.3 Starting without installation

This section is UNIX only!

It is extremely valuable to start the editor without installing it. You can do so by using the *runwrapper.sh* script in the repository. This script will spawn a new shell that will have environment set so that CEED finds its own modules and PyCEGUI. By default it assumes the following directory structure:

```
$prefix/CEED/bin/runwrapper.sh
$prefix/cegui_mk2/build/lib/PyCEGUI.so
```

If your directory structure looks differently you need to alter the script.

Chapter 9

Directory structure

9.1 Top directory

9.1.1 maintenance script

Provides means to compile Qt .ui files, build documentation, fetch newest CEGUI datafiles and make a tarball for CEED releases.

maintenance-temp is a directory with various temporary data that maintenance script needs to run.

9.1.2 perform-pylint

Runs *pylint* over the codebase, results will be stored in pylint-output. It is imperative to run this script, especially before releases, it often uncovers nasty bugs. Even though *pyflakes* has no helper script to run it, you can run it as well, there are no configuration or such files required.

9.1.3 setup.py

Used to install CEED system-wide. Running python setup.py install as root will get the job done. Make sure you already have all the dependencies installed.

Can also be used to create tarballs, the maintenance script may be better for that though, see Section 9.1.1.

9.1.4 cx_Freezer.py

This is a setup.py script that is adapted for freezing the application into a bundle using cx_Freeze. The resulting bundle does not need any dependencies,

not even *Python*. Tested on Windows 7 and GNU/Linux distros, both 32bit and 64bit.

Might need copying of some dependencies the script fails to pick up!

Please see the cx_Freeze documentation [22] for more information.

9.1.5 copyright related

Also includes the *AUTHORS* file with CEED contributors and several *COPYING* files of libraries we bundle in Windows and MacOS X builds.

9.2 bin directory

All contents are executable, these are entry points to various functionality of CEED.

9.2.1 ceed-gui

Starts the CEED interface. Provides several CLI options that may be very useful for development, especially auto opening of projects and files after start, see ./ceed-gui –help.

9.2.2 ceed-mic

This is the CLI metaimageset compiler, see the *User manual* for more info.

9.2.3 ceed-migrate

CLI interface to the compatibility machinery in CEED, can be useful for testing newly developed layers, see ./ceed-migrate –help for more info.

9.2.4 runwrapper.sh

Can be used to start CEED without having to install it, see Section 8.3 for more info.

9.3 build directory

Contains results of cx_Freeze build process, see Section 9.1.4 for more info.

9.4 ceed directory

This is where the bulk of the codebase resides. The directory is a *Python package* and none of its files should be executable.

9.4.1 action subpackage

Implements the Action API and defines basic global actions.

9.4.2 cegui subpackage

Wraps Embedded CEGUI (see Section 10.6 for more details). Also provides base classes for CEGUI widget manipulators and all the machinery that they require - GraphicsScene, GraphicsView, ...

9.4.3 compatibility subpackage

Implements the Compatibility API, contains implementations of all the stock *Type Detectors* and *Compatibility Layers*.

9.4.4 editors subpackage

This subpackage encapsulates all editing functionality within CEED. All classes that inherit from TabbedEditor except the convenience wrapper classes should be implemented inside this subpackage.

You can find implementation of imageset editing in the *imageset* subpackage, layout editing in the *layout* subpackage, ...

9.4.5 metaimageset subpackage

Classes required for metaimageset parsing, saving and compiling are implemented in this subpackage. This is what ceed-mic (see Section 9.2.2) uses internally to compile a metaimageset.

9.4.6 propertytree subpackage

UI to inspect and change properties of any class inheriting *CEGUI::PropertySet*.

9.4.7 settings subpackage

Implements the Settings API, defines basic global settings entries.

9.4.8 ui subpackage

Contains .ui files created using *Qt Designer*. The maintenance script is used to compile these into *Python modules*. See Section 10.9 for more info.

9.5 data directory

Contains icons, the splashcreen, stock property mappings, sample CEGUI datafiles and sample project files.

9.6 doc directory

Contains LyX source code for developer manual, quickstart guide and user manual. Also contains the PDF versions after ./maintenance build-docs has been executed (see Section 9.1.1).

Chapter 10

Core API

The whole code is divided into folders where the root folder provides basic reusable functionality (project management, undo view, tab management, ...) and the editors themselves are providing editing facilities for various file types.

10.1 TabbedEditor

A base class for editors hosted in a tab. If you are writing new editing functionality for CEED you definitely need to inherit from this class.

10.1.1 Responsibilities

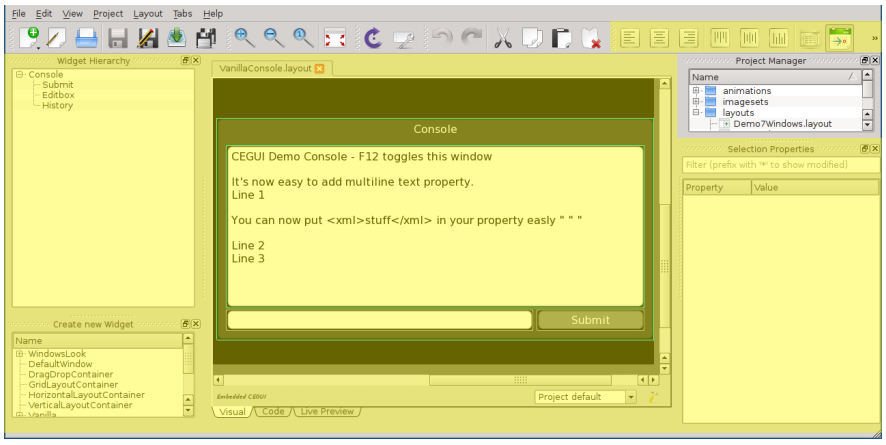


Figure 10.1: tabbed editor responsibilities are highlighted in yellow

The most important part of a TabbedEditor is its widget. The widget represents the central part in Figure 10.1. TabbedEditors also often add toolbars, dock widgets and other elements.

10.1.2 Life cycle

Each tabbed editor goes through the following cycle:

1. Construction of the class
2. Initialisation
    (a) all the supporting widgets get created
    (b) the file is loaded and processed
3. Activation
    (a) this puts the tabbed editor "on stage"
4. User interaction
5. Deactivation
6. Finalisation
    (a) the editor is no longer shown in the interface
7. Destruction
    (a) all held data and widgets are destructed

10.1.3 Derived classes

To avoid repeating code and adhere to the DRY principle [15], there are 2 very important classes that add functionality to TabbedEditor that you want to inherit if applicable to avoid reinventing.

UndoStackTabbedEditor

Very useful in case you are already using the Qt's UndoStack. This connects all the necessary calls and exposes undo and redo of the undo stack to the rest of the application.

MultiModeTabbedEditor

Useful when you want multiple editing modes. As an example, let us take the layout editor. It has three modes - visual, code and live preview. You can freely switch between them and they each offer a different look at the same data. At any point in time you are viewing/editing in one mode only. Please note that you must be using UndoStack in this situation as switching modes is an undo action.

Each mode has its own life cycle and depends on the life cycle of its host tabbed editor. First the tabbed editor gets on "the stage" and then the editor's mode is asked to activate itself.

```
# the host tabbed editor gets constructed and activated
A.deactivate()
B.activate()
# the user merrily edits in the B edit mode
```

Figure 10.2: process of switching from edit mode A to B

The actual mode switch process is a bit more involved because of the necessity to make mode switch an undoable action. You can see the full implementation of it in *ceed.editors.multi.MultiModeTabbedEditor.slot_currentChanged*.

10.2 Undo / Redo

One of the cornerstones of CEED is the ability to undo everything. This is implemented using Qt's QUndoCommand class. Each TabbedEditor has its own independent undo stack, undo commands are never shared across editors.

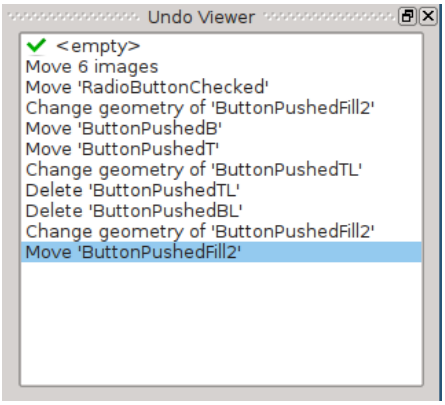


Figure 10.3: example of an undo stack

10.2.1 Principles

- everything that changes data has to be an UndoCommand
- all data that undo command stores in itself must be "independent", storing references to widgets would not work if there is a Destroy-Command that invalidates them

- state switching that would make some undo commands not applicable have to be undo commands themselves

10.2.2 Moving in the undo stack

Let us consider the undo stack shown in Figure 10.3. If user clicks the <*empty*> line, all the undo commands will get .undo() called in the bottom-up order. If now the user clicks the *Move 'ButtonPushedFill2'* line again, the commands will get .redo() called in the top-down order. It is important to notice that the undo commands are always acted upon sequentially and that order of the calls matter! Some of the commands might not even make any sense if they are called out of order. Consider a *Create Image 'XYZ'* command followed by *Move 'XYZ'*. They need to be acted upon in the right order otherwise the Move command is asked to move a non-existent image.

10.3 Property editing

A lot of CEGUI classes provide basic introspection via property strings. CEED has a set of classes to reuse when you want to edit properties of widgets or any other classes that inherit from PropertySet.

10.3.1 Usage

Even though the *propertytree* subpackage (see Section 9.4.6) gives you access to its very internals and allows very advanced uses, including using it on classes that do not even inherit from the CEGUI::PropertySet, only the basic usage scenarios will be discussed in this document.

```
from ceed import propertysetinspector
from ceed import mainwindow

# parent is a QWidget and can be None
inspector = propertysetinspector.
   PropertyInspectorWidget(parent)
self.inspector.ptree.setupRegistry(propertytree.editors
   .PropertyEditorRegistry(True)
pmap = mainwindow.MainWindow.instance.project.
   propertyMap
self.inspector.setPropertyManager(propertysetinspector.
   CEGUIPropertyManager(pmap))
```

Figure 10.4: creating a property inspector widget

```
# inspector is a property inspector as created
   previously

inspector.setPropertySets([propertySetToInspect])
```

Figure 10.5: inspecting a PropertySet using a property inspector

10.4 Settings API

Whenever you want users to be able to change some value to affect behavior of the application, consider using the Settings API. You only need to define the *settings entry* and the UI that allows changing it will be auto-generated for you.

```
category = settings.createCategory(name = "layout",
   label = "Layout editing")

visual = category.createSection(name = "visual", label
   = "Visual editing")

visual.createEntry(name = "continuous_rendering",
                   type = bool,
                   label = "Continuous rendering",
                   help = "Check this if you are
                      experiencing redraw issues...",
                   defaultValue = False, widgetHint = "
                      checkbox",
                   sortingWeight = -1
```

Figure 10.6: defining a settings entry

It is recommended to query the settings entry once and keep the reference stored to avoid having to look it up frequently.

```
entry = settings.getEntry("layout/visual/
    continuous_rendering")
# entry is a reference to SettingsEntry class
# we get the fresh value whenever we use entry.value
    later in the code
print("Continuous rendering is %s" % ("on" if entry.
    value else "off"))
```

Figure 10.7: using a settings entry

10.5 Action API

Whenever there is an action needed you are advised to use the action API, see ceed.action module. The actions inherit from QAction and offer the same functionality but shortcuts are handled automatically for the developer, including UI for the user to remap them.

To use the Action API you have to define your actions first, this is usually done in a separate file to keep things clean. See *editors/imageset/action_decl.py* and *editors/layout/action_decl.py*. Then you query for this action in your code and connect your signals to it. You can use the convenience ConnectionMap to ease mass connects and disconnects.

```
cat.createAction(
              name = "align_hleft",
              label = "Align &Left (horizontally)",
              help = "Sets horizontal alignment of all
                 selected widgets to left.",
              icon = QtGui.QIcon("icons/layout_editing/
                 align_hleft.png"))
```

```
cat.createAction(
              name = "snap_grid",
              label = "Snap to &Grid",
              help = "When resizing and moving widgets,
                 if checked this makes sure...",
              icon = QtGui.QIcon("icons/layout_editing/
                 snap_grid.png"),
              defaultShortcut = QtGui.QKeySequence(
                 QtCore.Qt.Key_Space)).setCheckable(True
                 )
```

Figure 10.8: defining new actions

You can check the shortcut remap UI generated for you in *Settings » Shortcuts*.

10.6 Embedded CEGUI

To make sure everything is rendered exactly as it will appear in CEGUI it is used in the editor. This also ensures that whatever custom assets you have, they will be usable in the editor exactly as they are in CEGUI itself.

10.6.1 PyCEGUI bindings

As CEGUI is a C++ library, making it accessible from Python is not trivial. I have written python bindings for CEGUI called PyCEGUI using *py++* and *boost::python* for this purpose. It is important to realise though that even though I tried to make it pythonic and reasonably safe, mistreating PyCEGUI can still cause segfaults and other phenomena usually prevented by using a scripting language.

10.6.2 Shared CEGUI instance

There is only one CEGUI instance in CEED. This makes tabbed editor switches slightly slower but CEED uses less memory. The main reason for this design

decision is that CEGUI did not have multiple GUI contexts at the time CEED was being designed.

Furthermore, the shared instance is wrapped in a "container widget" which provides convenience wrappers. That way developer can avoid dealing with *OpenGL* and *QGLWidget* directly.

```
ceguiContainerWidget = mainwindow.MainWindow.instance.
    ceguiContainerWidget

# parentWidget is the widget that will host the CEGUI
    rendering, it cannot be None!
ceguiContainerWidget.activate(parentWidget, self.scene)
ceguiContainerWidget.setViewFeatures(wheelZoom = True,
    continuousRendering = True)

# you can then use CEGUI directly through PyCEGUI, the
    result will be rendered
# to the host widget specified previously
PyCEGUI.System.getSingleton().getDefaultGUIContext().
    setRootWindow(self.rootPreviewWidget)

# ... rendering, interaction, etc.

# after your work is done, deactivate the container
    widget
ceguiContainerWidget.deactivate(self.ceguiPreview)
```

Figure 10.9: accessing and using the CEGUI instance

Always clean up!

The *CEGUI* container widget is shared, therefore the whole *CEGUI* instance and the default *GUIContext* are shared. *CEGUI* resources are not garbage collected, they are created in the C++ world and have to have their life cycles managed manually. Make sure you always destroy all your widgets and other resources after use. They will not get cleaned up until the whole editor is closed!

Beware of name clashes!

Becuase the CEGUI instance is shared there can be name clashes for many resources - images, animation definitions, ... A good way to circumvent this is to generate unique games with an integer suffix and hide the fact from the user.

This is what the *Animation list editor* does internally, for more details see *ceed.editors.animation_list*.

10.7 Compatibility layers

Compatibility is only dealt with on data level. The editor itself only supports one version of each format and layers allow to convert this raw data to other formats. Here is an example of how to do that:

```
# we want to migrate and imageset from data format "foo
    " to "bar"
# data is a string containing imageset in "foo" format

from ceed.compatibility import imageset as compat
convertedData = compat.manager.transform("foo", "bar",
    data)
```

There are also facilities to guess types of arbitrary data. See API reference of CompatibilityManager for more info.

10.7.1 Testing compatibility layers

Running the GUI and loading files manually by clicking is not practical for compatibility layer development and testing. Use the *ceed-migrate* executable instead. See Section 9.2.3.

10.8 Model View (Controller)

As most editing applications we have the MVC paradigm [18]. When I say something is the *model* I mean that it encapsulates and contains the data we are editing. The *view* on the other hand encapsulates the facility to view the data we are editing in their current state. The *controller* allows the user to interact with the data. Most of the time *view* meshes with *controller* as it does in the Qt world so we are using one class instance for both *view* and *control*.

Separating model from view helps make the code more maintainable and cleaner. It also makes undo command implementation easier.

10.9 Qt designer .ui files

Qt designer allows RAD so it pays off to keep as much GUI layout in .ui files as possible. Whenever you are creating a new interface, consider creating it with the Qt designer instead of coding it manually.

10.9.1 Compiling

The files have to be compiled into *Python modules*.

Development mode

The preferred method if you want to continuously develop CEED. Allows automatic recompilation of all ui files.

```
$ vim ceed/version.py
# make sure the DEVELOPER_MODE line is set to True
```

Figure 10.10: turning the developer mode on

maintenance script

If you only want to compile the ui files rarely you are better off with the maintenance script. See Section 9.1.1.

```
./maintenance compile-ui-files
```

Figure 10.11: recompiling ui files via the maintenance script

Chapter 11

Editing implementation

11.1 Imageset editing

Lives in the *ceed.editors.imageset* package. Provides editing functionality for CEGUI imagesets. Please see the CEGUI imageset format documentation [17] for more details about the format.

11.1.1 Data model

Classes from the *ceed.editors.imageset.elements* package are used to model the data instead of using CEGUI in this editor. The reason is relative simplicity of the data and big changes to the image API between CEGUI 0.7 and 1.0. Compatibility layers are used to convert given data to the native format before they are loaded into the data model. See Section 10.7 for more details.

11.1.2 Undo data

Undo data are implemented using string for image definition reference and Python's builtin types to remember geometry.

11.1.3 Multiple modes

It is a multi-mode editor with visual and code modes. The code mode always uses and displays native CEGUI 1.0 data.

11.1.4 Copy / Paste

Copy paste is implemented using custom MIME type and bytestreams. It is even possible to copy image definitions across editor instances.

11.2 Layout editing

Located in the *ceed.editors.layout* package. CEGUI Window is used to model the entire layout hierarchy. We use WidgetManipulator class to add serialisation (for undo/redo), resizing handles and more to windows. It is a multimode editor with visual, code and live preview modes. The live preview mode does no editing, instead it just views the current layout and allows user to interact with it to test it.

11.2.1 Data model

Layout editing operates of widget hiearchies, a data model natively implemented in CEGUI that we use directly. Since CEGUI does not have global window names since version 1.0 we do not even have to worry about name clashes.

11.2.2 Undo data

Undo data are implemented using strings for widget path reference and widget properties are serialised using Python's builtin types.

LookNFeel property caveat

When you change the LookNFeel property the auto child widgets get destroyed and constructed anew. This breaks undo history and is not allowed at the moment. I don't it is worth the effort to support this. Either way we would have to "alter history" in some cases. Changing it in code mode will of course work because the entire hierarchy will be reconstructed from scratch.

WindowRenderer property caveat

Similar to the LookNFeel case it makes changes to the window that break undo history. Right now it is disallowed to change it from the editor. Changing it in code mode will of course work because the entire hierarchy will be reconstructed from scratch.

11.2.3 Multiple modes

Visual, Code and *Live preview* modes are provided. Code is a simple XML editing mode but the other two are implemented using embedded CEGUI.

11.2.4 Copy / Paste

Copy paste is implemented using custom MIME type and bytestreams. It is even possible to copy widget hierarchies across editor instances.

11.3 Animation editing

Located in *ceed.editors.animation_list* package. We use wrappers to deal with the fact that CEGUI has no model for a list of animations.

KeyFrames had to have indices added because comparing floats for equality is unreliable. So in the end we sort all keyframes by position and figure out their indices from that. To avoid placing two keyframes at the exact same position we add a small epsilon until we have no clashes whenever we encounter this possibility.

Chapter 12

Contributing

12.1 Coding style

CEED does not follow the *PEP8* style recommendation when it comes to method and variable naming. The reason I chose to use camelCase for methods and variables is that PySide and CEGUI both use that and CEED calls a lot of methods form these 2 APIs. The code looked much better with camelCase naming.

Use the following rules for all contributed code to CEED:

- use 4 spaces for indentation
- use CamelCase for class naming
- do not use wildcard imports [1]
- use camelCase for method and variable naming
- document methods and classes with the triple quote docstyle syntax
- comment all other things with # prefix only

12.2 Communication channels

You can reach the CEGUI team using:

- IRC: #cegui on irc.freenode.net[2]
- email: team@cegui.org.uk

[1] `from package import *` cannot appear anywhere in the code.
[2] See `http://freenode.net` for more information about the network.

12.3 DVCS - forking

Create a fork of *http://crayzedsgui.hg.sourceforge.net:8000/hgroot/crayzedsgui/CEED* on `http://bitbucket.org` or elsewhere. Start each feature or substantial fix in a separate branch, this makes it easy to review and possibly reject some parts without rejecting everything. When you are finished with your branch make sure you merge all upstream changes if any. Having to deal with merge conflicts makes the reviewers more likely to postpone integration. After all of this is done, simply contact upstream developer to merge your changes into the main repository. You can usually reach someone through IRC (freenode/#cegui), mantis bug tracker or email (*team@cegui.org.uk*).

12.4 The old fashioned way - patches

You can alternatively just send unified diff patches by email if you so desire. Use the *team@cegui.org.uk* email address. Make sure you state what the patch-set is based on.

Part V

Conclusion

Chapter 13

Statistics and graphs

13.1 Adoption

Overall I think CEED has been fairly successful with roughly 2500 downloads in total [2] as of 22nd July 2012. Three professional proprietary game development studios reported that they were using it and were a valuable source of bug reports and feedback. It has also been packaged for ArchLinux community repository [1], I sadly have no statistics of usage from this source.

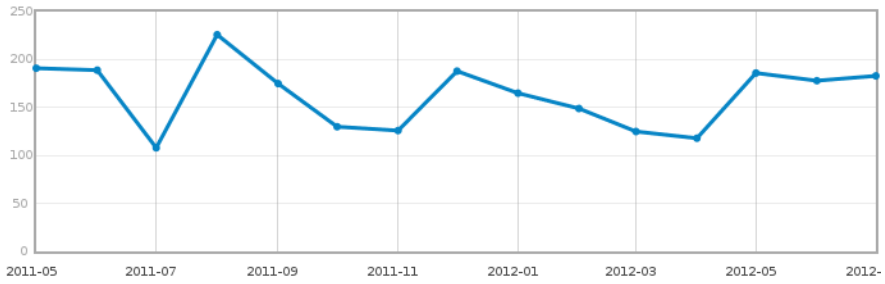


Figure 13.1: number of downloads per month on SourceForge [2]

The download graph does not show any increasing trend which suggests that unstable editing application does not convince new users to migrate to CEGUI, the pool of potential users remains more or less constant.

Platforms of users

Windows has been very decisively the most common platform of CEED users, 88% of all users [2] downloading CEED from SourceForge had Windows. This is not surprising as the target audience are artists and Windows is the market leader in desktop operating systems at the moment.

GNU/Linux took second place, surprisingly miles ahead of Macintosh. The most likely explanation is that MacOS X standalone builds of CEED were not provided before snapshot8 release. The friction of building all the dependencies probably prevented users from trying the application.

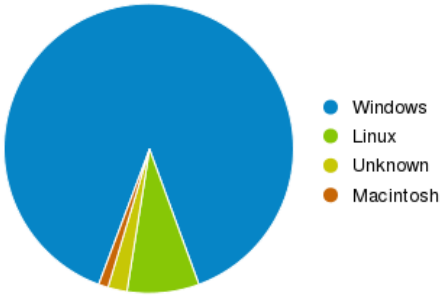


Figure 13.2: platforms of CEED users

Countries of users

China is dominating the chart with 746 CEED downloads in total [2] as of 22nd July 2012. This is very surprising to me, I have not heard from any Chinese user as far as I know. I do not think it is some sort of a measuring mistake because I know of many projects in China that use CEGUI, yet we never hear from the developers. The most probable explanation is that there are cultural and language barriers that prevent users from interaction with the upstream.

United States took second place with 341 downloads in total, Germany took third with 208 downloads in total.

13.2 Development

13.2.1 Timeline

My goal was to proceed at a steady pace, focusing on bug reports and issues. The graph in Figure 13.3 shows that there were next to no drastic rewrites and development proceeded at a predictable rate. The last four months of development were focused on integration testing and bug fixing. Very few features were added during that time.

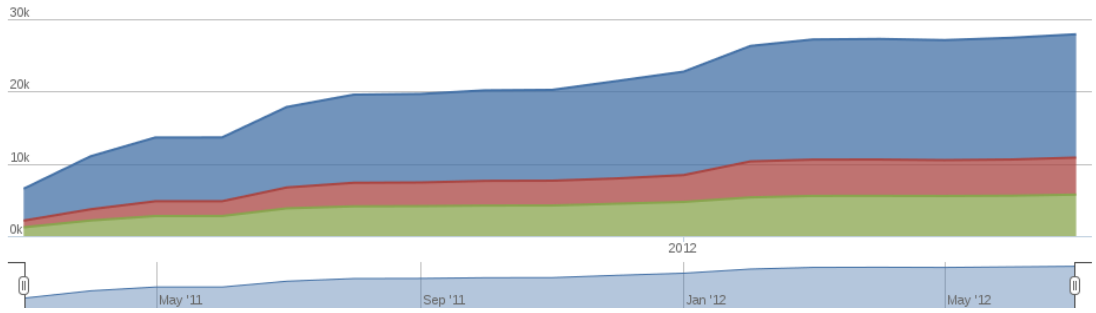


Figure 13.3: lines of code timeline (blue: code, red: comments, green: blank space) [3]

The graph in Figure 13.4 shows a few alarming spikes in development. I believe some of them were caused by me merging community contributions, several of which were tens of commits at a time. Brief pauses of development were also caused by time constrains.

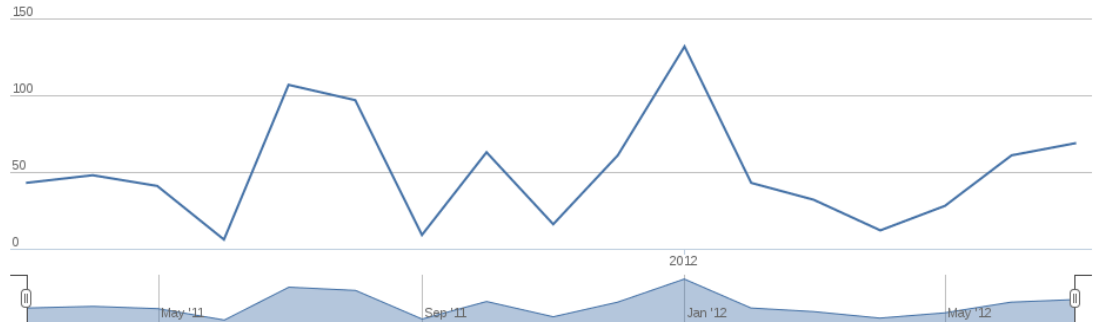


Figure 13.4: commits per month [3]

13.2.2 Contributors

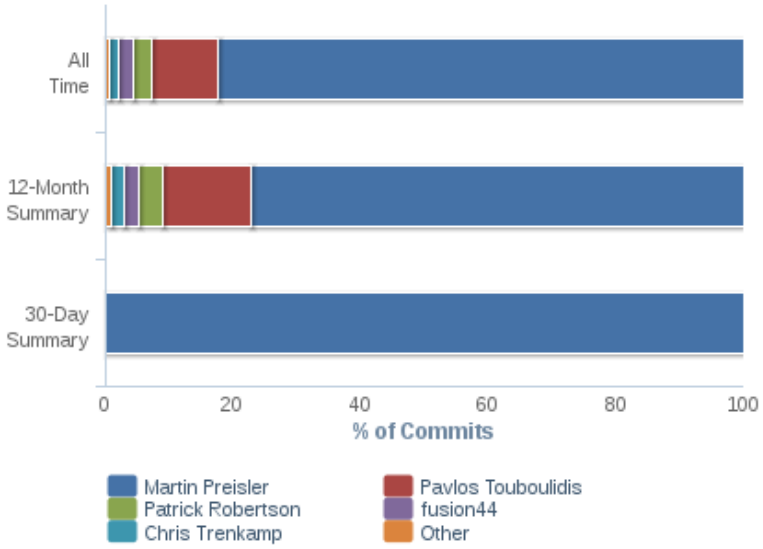


Figure 13.5: contributor commit breakdown including Martin Preisler [3]

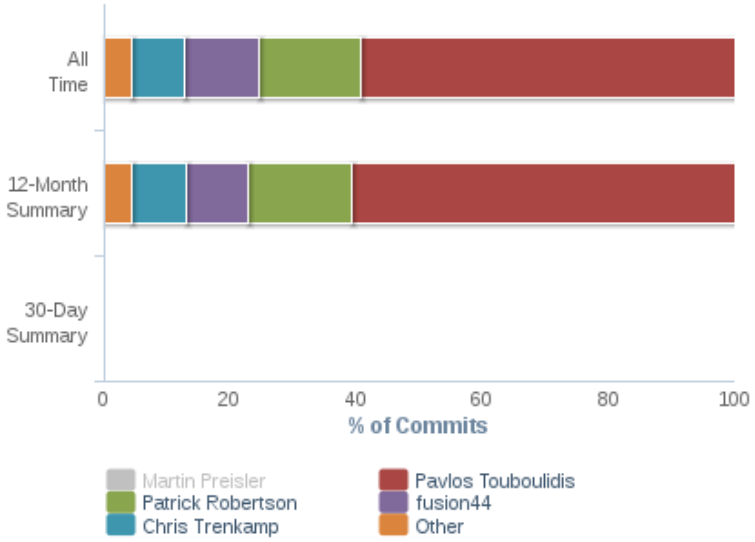


Figure 13.6: contributor commit breakdown excluding Martin Preisler [3]

Despite me providing a decisive majority of code, the entire application was a collaboration of many authors around the world.

We can see that external contributors were submitting commits with a slightly increasing trend. Four developers account for a staggering 99% of all commits. We have to consider the fact that merging and branch closing commits were done solely by Martin Preisler, so the percentage of his contribution might be a couple per cent lower.

13.3 Codebase

Consists of 27,874 lines of code [3] as of 22nd July 2012. 17,036 are code lines, 5,110 are comment lines and 5,728 are blank. The COCOMO model estimates 4 person-years of effort and values the project at a generous $210,759 with programmer annual salary of $55,000 [3]. It is important to note that the COCOMO model typically overvalues young projects [16].

13.4 Issue tracking

As of 22nd July 2012 a total of 291 issues have been reported. 205 of them are marked as resolved. The most common category of issues is the "General" category with 114 issues reported. Layout editing takes second place with 91 issues. [4]

Chapter 14

Future development

14.1 Unfinished features

Animation editing

Some of animation editing is implemented but I got sidetracked with all the issues reported and could not finish it in time. There is a lot of user interface work to do there but the core basics are implemented.

Better shortcut implementation

Shortcuts are application-wide at the moment, meaning they are not tied to a widget and do not need focus in a particular widget. This leads to various issues like CTRL+S saving to a file and moving currently selected image definition down in imageset editing. This should be fixed by requiring shortcuts to be bound to a widget and only triggered when such widget has keyboard focus.

Support for TabControl in layout editor

Despite there being support for AutoWindows in general, TabControl can still cause CEED to crash and is generally not supported.

14.2 Software is never truly finished

I expect to get even more users with its first stable release that shall happen after CEGUI 1.0 release. Many teams are worried about using it as part of their pipeline when it has not been declared stable yet.

As it is free software I am expecting more contributors and more features in the future. There is still much to implement and I do not think CEED will ever be proclaimed as finished.

Nomenclature

CEGUI	CrazyEddie's GUI System
EULA	End User License Agreement
Falagard	Name of CEGUI's skinning system
GPU	Graphics Processing Unit
GUI	Graphical User Interface
LookNFeel	CEGUI term for widget skinning data
NDA	non-disclosure agreement
OOP	Object Oriented Programming
RAD	Rapid Application Development
TTF	True Type Font
UDim	Unified Dimension
UI	User Interface
WYSIWYG	What You See Is What You Get

Bibliography

[1] AUR CEED page. URL: http://aur.archlinux.org/packages.php?ID=60094.

[2] CEED SourceForge download page, . URL: http://sourceforge.net/projects/crayzedsgui/files/CEED/.

[3] Ohloh page for CEED, . URL: http://www.ohloh.net/p/CEED.

[4] CEGUI mantis tracker, . URL: http://cegui.org.uk/mantis.

[5] CEGUI website, . URL: http://cegui.org.uk.

[6] GIMP rectangle selection tool documentation. URL: http://docs.gimp.org/en/gimp-tool-rect-select.html.

[7] Mantis website. URL: http://www.mantisbt.org/.

[8] About page on the MyGUI website. URL: http://mygui.info/#about.

[9] Ogre3D website. URL: http://ogre3d.org.

[10] The original CEED forum thread. URL: http://cegui.org.uk/phpBB2/viewtopic.php?f=15&t=5318.

[11] PySide website. URL: http://www.pyside.org/.

[12] ropevim website. URL: http://rope.sourceforge.net/ropevim.html.

[13] Summoning Wars website. URL: http://sumwars.org.

[14] Worldforge website. URL: http://worldforge.org.

[15] Andrei Alexandrescu and Herb Sutter. *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*. 2004. ISBN 978-0321113580.

[16] Barry W. Boehm. *Software Cost Estimation with COCOMO II*. 2000. ISBN 978-0137025763.

[17] CEGUI development team. Imageset xml format specification. http://www.cegui.org.uk/docs/current/xml_imageset.html.

[18] Alan Ezust. *An Introduction to Design Patterns in C++ with Qt 4*. 2006. ISBN 978-8131713266.

[19] Martin Fleurent. AUTHORS file in ceimageseteditor tarball. URL: http://prdownloads.sourceforge.net/crayzedsgui/CEImagesetEditor-0.7.1.tar.gz?download.

[20] Patrick Kooman. AUTHORS.txt in celayouteditor tarball. URL: http://prdownloads.sourceforge.net/crayzedsgui/CELayoutEditor-0.7.1.tar.gz?download.

[21] Mark Summerfield. *Rapid GUI Programming with Python and QT: The Definitive Guide to PyQt Programming*. 2007. ISBN 978-0132354189.

[22] Anthony Tuininga. cxFreeze documentation. http://cx_freeze.readthedocs.org/en/latest/index.html.

Appendix A

CD attachment

A.1 License information

Files stored on the CD are freely redistributable. See *COPYING* and **-COPYING* files on the CD for more info.

A.2 Directory structure

README.txt information about contents of the CD

COPYING licensing of CEED

***-COPYING** licensing of libraries bundled with standalone builds

ceed-thesis.pdf digital form of this document

win32/ Windows standalone build of the application

osx/ MacOS X standalone build of the application

src/CEED/ source code of CEED including Mercurial data with full history

src/CEED/doc/user-manual-src LyX sources for the user manual

src/CEED/doc/developer-manual-src LyX sources for the developer manual

src/cegui_mk2/ MIT-licensed source code of CEGUI used to compile the builds

src/ceed-thesis/ LyX sources of the thesis itself, figures' imagery